DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

Optimización de filtros usados para detección de bordes a través del uso de instrucciones SIMD

22 de agosto de 2013 — Organización del Computador II

Integrante	LU	Correo electrónico
Artuso, Pablo Agustín	282/11	p.artu@hotmail.com
Landini, Federico Nicolás	034/11	federico91_fnl@yahoo.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Los algoritmos de detección de bordes son muy utilizados en la práctica ya que resultan un paso previo necesario en toda aplicación que involucre reconocimiento de objetos o formas. En este trabajo presentamos cuatro algoritmos de detección de bordes: Roberts Cross, Sobel, Prewitt y Canny; presentando cada uno de ellos ciertas características que analizaremos. Para cada uno de ellos proponemos una implementación en lenguaje C y otra en lenguaje assembly (utilizando instrucciones del tipo SSE) a fin de poder comparar los rendimientos en cada caso. Dado que los algoritmos presentan distintas técnicas también se hará un análisis de las capacidades de cada uno frente al costo en términos de tiempo que exigen.

Además de los algoritmos se provee una interfaz gráfica que permite al usuario elegir entre distintos archivos para aplicar sobre ellos los distintos algoritmos de detección de bordes. De este modo se pueden ver los bordes en imágenes, videos e incluso sobre lo capturado en tiempo real por una webcam. Teniendo en cuenta estas posibilidades se provee en última instancia una herramienta capaz de poder medir la distancia entre dos puntos de la imagen seleccionados por el usuario a partir de una escala fijada. La ventaja que proporciona esta herramienta es poder medir un determinado objeto incluso en tiempo real.

2. Desarrollo

2.1. Operador Roberts Cross

El algoritmo de Roberts Cross fue diseñado en 1963 por Lawrence Roberts. La idea de su funcionamiento es aproximar el gradiente de la imagen para cada punto; es decir, cuánto cambia la intensidad entre píxeles adyacentes. Dado que hay grandes cambios en las intensidades entre píxeles adyacentes de la imagen si y sólo si se presentan bordes, valores grandes para el gradiente implican la existencia de un borde. Dado que se opera de manera discreta y se trabaja en una imagen con píxeles, la técnica propuesta por Roberts consistió en tomar cuatro píxeles adyacentes, calcular las componentes del gradiente en dos direcciones (las diagonales) y luego tomar como valor del mismo la norma del vector bidimensional formado por dichas componentes.[1]

Para poder obtener el gradiente sobre el píxel $I(i,j)$ de la imagen en las direcciones diagonales se consideran las siguientes dos convoluciones:

$$G_x(i,j) = \begin{bmatrix} +1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \text{y} \qquad G_y(i,j) = \begin{bmatrix} 0 & +1 \\ -1 & 0 \end{bmatrix}$$

Consideremos una porción de la imagen con los siguientes nueve píxeles:

$$\begin{array}{cccc} I(i,j) & I(i+1,j) & I(i+2,j) & \dots \\ I(i,j+1) & I(i+1,j+1) & I(i+2,j+1) & \dots \\ I(i,j+2) & I(i+1,j+2) & I(i+2,j+2) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

Entonces la aplicación de las convoluciones nombradas anteriormente en el píxel (i,h) resultan en $G_x(i,j) = I(i,j) - I(i+1,j+1)$ y $G_y(i,j) = I(i+1,j) - I(i,j+1)$. Sucede entonces que este filtro no se puede aplicar en los píxeles de la última fila ni en los de la última columna; en esos casos se procede a determinar que los resultados de ambas convoluciones son 0.

Una vez calculados $G_x(i,j)$ y $G_y(i,j)$, se determina $|\nabla I(i,j)| = G(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}$ que es elSe puede obtener, además, el ángulo de dicho gradiente como $\Theta(i,j) = \arctan(\frac{G_y(i,j)}{G_x(i,j)})$.

2.1.1. Algoritmo en C

El algoritmo resulta bastante simple pues expresa exactamente lo explicado antes. El siguiente pseudocódigo explicita la forma en la que se opera:

Algoritmo 1 Roberts Cross en C

Aclaraciones: $*src$ es un puntero a la imagen fuente, $*dst$ es un puntero a la imagen destino, m y n son alto y ancho de la imagen en cantidad de píxeles respectivamente, row_size hace referencia (tanto para src omo para dst) al ancho de la imagen incluyendo el posible padding.

```
1:  procedure RobertsCross c(*src, *dst, m, n, src_row_size, dst_row_size)
2:      ancho = 0
3:      alto = 0
4:                                  ▷ Los ciclos llegan hasta las anteúltimas fila y columna pues éstas no se procesan
5:      while alto < m − 1 do
6:          while ancho < n − 1 do
7:              gx = *src − *(src + 1 + src_row_size)
8:              gy = *(src + 1) − *(src + src_row_size)
9:              *dst = √(gx * gx + gy * gy)
10:             src = src + 1
11:             dst = dst + 1
12:             ancho = ancho + 1
13:         *(dst − 1) = 0                  ▷ Fija los valores de la última columna como 0
14:         src = src + src_row_size − n
15:         dst = dst + dst_row_size − n
16:         ancho = 0
17:         alto = alto + 1
18:     dst = dst + dst_row_size − n
19:     ancho = 0
20:     while ancho < n do                  ▷ El ciclo fija los valores de la última fila como 0
21:         *dst = 0
22:         dst = dst + 1
23:         ancho = ancho + 1
24: end procedure
```

2.1.2. Algoritmo en Assembly

A fin de poder aprovechar las posibilidades que ofrece la tecnología SSE, el procesamiento de los píxeles en este lenguaje será distinto. Dado que por cada acceso a memoria se leen 16 bytes y para el procesamiento de cada uno se requiere conocer a aquellos a la derecha y abajo entonces necesariamente se deben leer 16 bytes de una fila y los 16 que ocupan las mismas posiciones en la fila inmediatamente inferior. Considerando que para cada píxel con el que se quiere trabajar se debe conocer al de la derecha, el décimosexto píxel leído no podrá ser procesado con la información obtenida en una lectura.

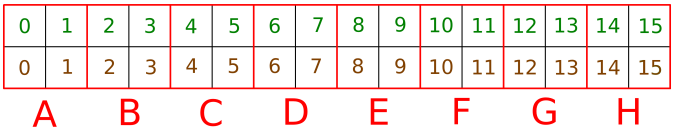
De este modo, cada vez que se levantan de memoria 16 bytes (y los 16 de la fila inferior) se procesan solamente los primeros 15. Es así entonces que la forma de recorrer la imagen será avanzando de a 15 píxeles sobre cada fila hasta llegar a un punto en el que la cantidad que queden por procesar sea menor a 16. Si la cantidad que queda es 1 entonces se terminó con la fila pues el último píxel no se procesa. Por otro lado, si queda una cantidad distinta entonces se vuelve hacia atrás por la misma fila la cantidad necesaria de píxeles para que resten 16 para llegar al final.

Como la escritura en memoria se hace también de a 16 bytes pero sólo se procesan los 15 primeros, entonces al último byte siempre se le da el valor 0 (píxel negro). Esto no genera ningún problema en el resultado de la fila pues se le da un valor negro al décimosexto píxel en cada escritura pero en la siguiente tirada de píxeles que se procesan se reescribe ese valor con el resultado correspondiente. Además, en el último caso hace que la última columna (que no se procesa sino que debe ser negra) ya tenga el valor 0. El siguiente código muestra este recorrido, obviando el código de procesamiento de los datos.

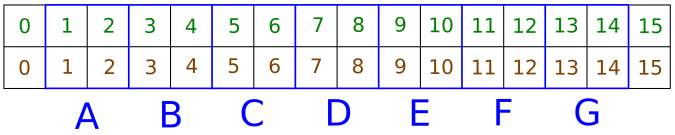
Aclaraciones: rdi tiene puntero a src, rsi tiene puntero a dst, rdx tiene filas, rcx tiene columnas, r8 tiene src_row_size, r9 tiene dst_row_size.

```
INICIO

.cicloPorFila:
        MOV r11, rcx                mueve la cantidad de columnas a r11
        MOV r14, rdi                mueve a r14 el puntero al primero de la fila
        MOV r13, rsi                mueve a r13 el puntero al dst
        CMP edx, 1                  compara con 1 pues la última fila no hay que procesarla
        JE .cicloPorColumnaUltima
.cicloPorColumna:
        CMP r11, 1                  ve si ya miró todas las columnas, la última no se procesa
        JE .cambiarDeFila
        CMP r11, 16                 ve si queda un tamaño menor al de procesado (o sea 16)
        JL .actualizarFinal

        ///* CODIGO *///

        ADD r14, 15                 avanza el iterador 15 bytes (cantidad procesada)
        ADD r13, 15                 avanza el iterador 15 bytes (cantidad procesada)
        SUB r11, 15                 resta 15 columnas (cantidad procesada)
        JMP .cicloPorColumna

.actualizarFinal:
        MOV r12, 16
        SUB r12, r11                r12 tiene lo que hay que retroceder para procesar 16
        ADD r11, r12                actualiza la cantidad de columnas
        SUB r14, r12                actualiza el src
        SUB r13, r12                actualiza el dst
        JMP .cicloPorColumna

.cambiarDeFila:
        ADD rdi, r8                 suma al src su row_size para ubicarlo en la sig. fila
        ADD rsi, r9                 suma al dst su row_size para ubicarlo en la sig. fila
        DEC edx                     resta una fila
        JMP .cicloPorFila

.cicloPorColumnaUltima:
        PXOR xmm15, xmm15           llena xmm15 de ceros
        CMP r11, 0                  compara para ver si no quedan más píxeles
        JE .salir
        CMP r11, 16                 ve si queda un tamaño menor al de procesado (o sea 16)
        JL .actualizarFinalUltima

        MOVDQU [r13], xmm15         imprime todo negro

        ADD r14, 16                 avanza el iterador 16 bytes (cantidad procesada)
        ADD r13, 16                 avanza el iterador 16 bytes (cantidad procesada)
        SUB r11, 16                 resta 16 columnas (cantidad procesadas en este caso)
        JMP .cicloPorColumnaUltima

.actualizarFinalUltima:
        MOV r12, 16
        SUB r12, r11                r12 tiene lo que hay que retroceder para procesar 16
        ADD r11, r12                actualiza la cantidad de columnas
        SUB r14, r12                actualiza el src
        SUB r13, r12                actualiza el dst
        JMP .cicloPorColumnaUltima

FIN
```

En cuanto al procesamiento de los píxeles fue dicho que se pueden procesar 15 con una lectura; sin embargo, a fin de lograr esto con sólo un acceso a memoria se deben trabajar los datos de manera adecuada. La decisión escogida en este sentido fue trabajar por una parte con los píxeles de posiciones pares (en la lectura) y por otro lado con aquellos de posiciones impares. Un esquema de cómo se caracterizan estas posiciones se encuentra en la Figura 1.



(a) En los recuadros rojos es donde se aplican las convoluciones para cada píxel de posición par.



(b) En los recuadros azules es donde se aplican las convoluciones para cada píxel de posición impar.

Figura 1: Elección de conjuntos de píxeles a procesar.

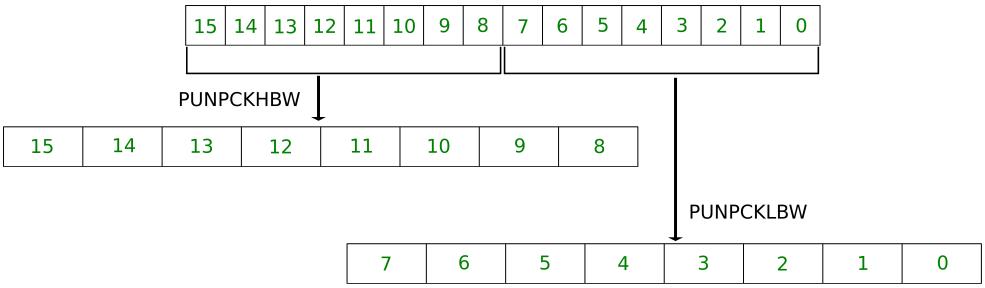
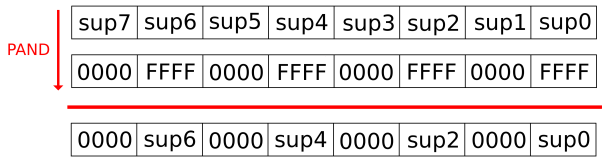


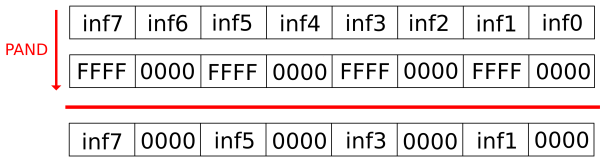
Figura 2: Extensión a words de los píxeles en un registro usando de por medio un registro con ceros. Esto se aplica a los 16 bytes de la fila superior y a los de la fila inferior guardando los resultados en xmm1, xmm2, xmm3 y xmm4.

El procedimiento que se realiza con los pares y los impares es muy similar, la diferencia es que para la segunda parte se toma lo leído de memoria (guardado en xmm1, xmm2, xmm3 y xmm4), se realiza un shift de cada uno de los registros por un espacio igual a 1 word y luego se realiza el mismo procedimiento que para los píxeles pares. Como se ve en la Figura 1, para los impares no existe un píxel procesado "H" y eso se debe justamente a que no se tiene información suficiente para trabajar con el décimosexto píxel.

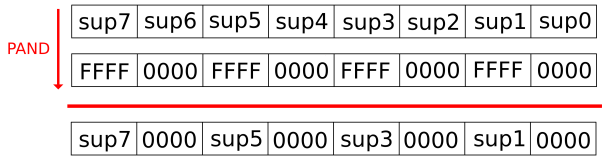
Inicialmente se leen de memoria 16 bytes de la fila que se procesa y los 16 bytes que se encuentran debajo. Puesto que se van a realizar ciertas operaciones con los datos y que cada uno de ellos tenga 1 byte no resulta suficiente se extienden los píxeles a word (Figura 2). Como se deben aplicar las convoluciones y éstas trabajan con sólo algunos de los píxeles entonces es preciso aplicar máscaras que dejen en los registros aquellos valores que interesan. La Figura 3 muestra cómo se seleccionan los valores al utilizar dichas máscaras.



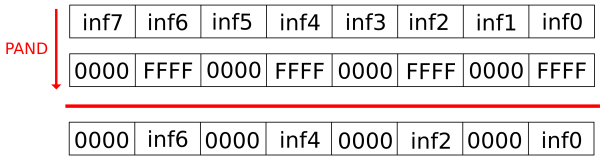
(a) Aplicación de la máscara sobre la parte baja de la fila superior para el cálculo de G_x. Con la parte alta se hace la misma operación.



(b) Aplicación de la máscara sobre la parte baja de la fila inferior para el cálculo de G_x. Con la parte alta se hace la misma operación.

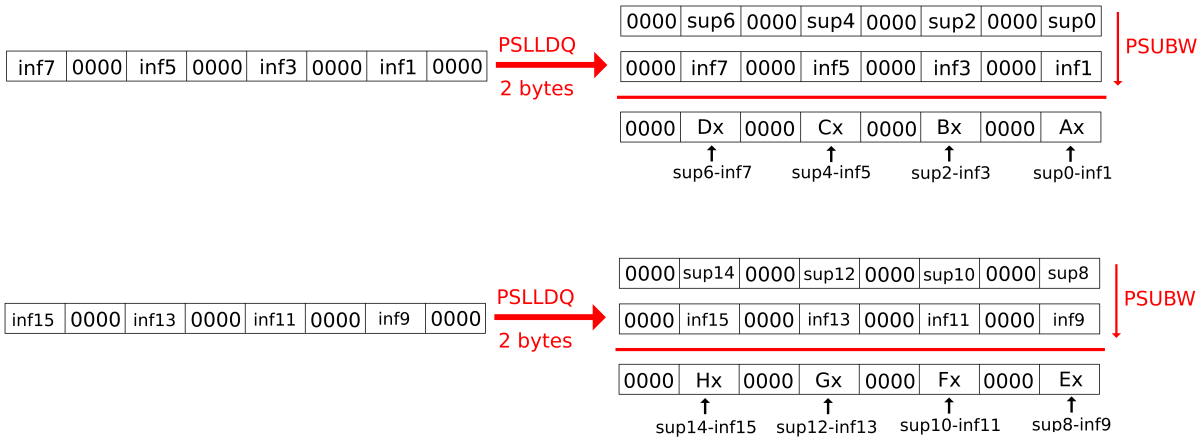


(c) Aplicación de la máscara sobre la parte baja de la fila superior para el cálculo de G_y. Con la parte alta se hace la misma operación.

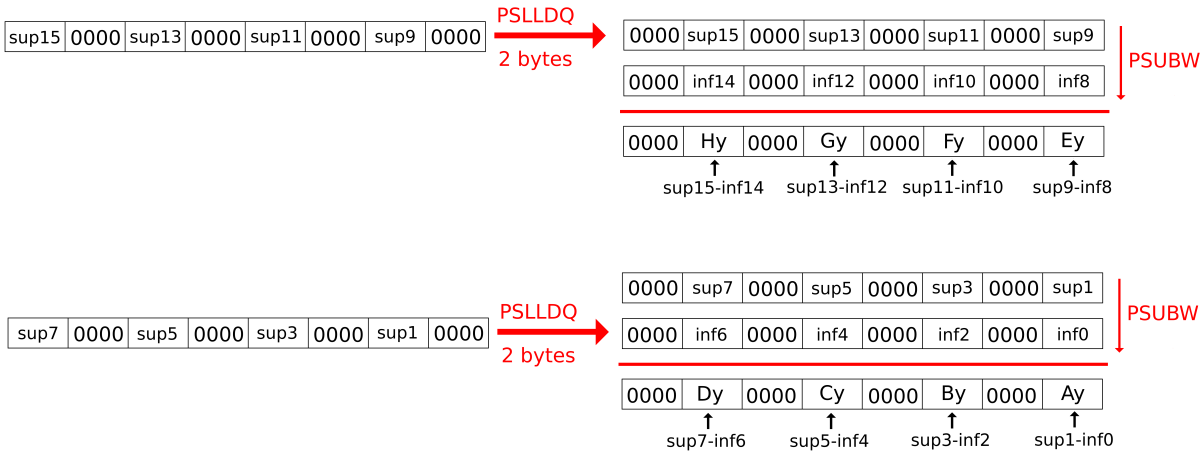


(d) Aplicación de la máscara sobre la parte baja de la fila inferior para el cálculo de G_y. Con la parte alta se hace la misma operación.

Figura 3: Selección de valores de píxeles con los cuales operar utilizando máscaras.



(a) Aplicación de shifts y resta para calcular G_x.



(b) Aplicación de shifts y resta para calcular G_y.

Figura 4: Cálculo de G_x y G_y.

Una vez obtenidos los cuatro registros después de aplicar las máscaras y a fin de poder calcular G_x y G_y, se deben aplicar shifts para poder restar los píxeles apropiadamente como se indica en la Figura 4.

Teniendo G_x y G_y resta elevarlos al cuadrado y luego calcular la raiz cuadrada de la suma de dichos cuadrados. Para elevar al cuadrado decidimos multiplicar al registro por sí mismo a través de las operaciones $PMULHW$ y $PMULLW$. Éstas devuelven en un registro la parte alta de la multiplicación y la baja respectivamente con lo cual es necesario luego otro paso para reconstruir el verdadero producto. Este procedimiento se detalla en la Figura 5. Una vez calculados los productos para todas las partes, deben sumarse correspondientemente.

El próximo paso consiste en obtener la raiz cuadrada de la suma anterior pero dado que esta operación se hace con números de punto flotante entonces se deben transformar los valores de entero a float, aplicar entonces la raiz cuadrada ($SQRTPS$) y volver a convertir los valores a entero (Figura 6). Una vez hecho esto, se obtendrán dos registros donde en cada uno de ellos habrá cuatro enteros de tamaño double word correspondientes al proceso completo de Roberts Cross en cuatro píxeles. Estos 8 píxeles procesados en total se corresponden con A, B, C, D, E, F, G y H de la Figura 1a. Se ve entonces que los píxeles no son consecutivos sino que se trata de los de posiciones pares por lo que entonces para dejarlos ubicados como tendrían que ir en la imagen destino hay que intercalar bytes con valor 0. El procedimiento se explica en la Figura 7a.

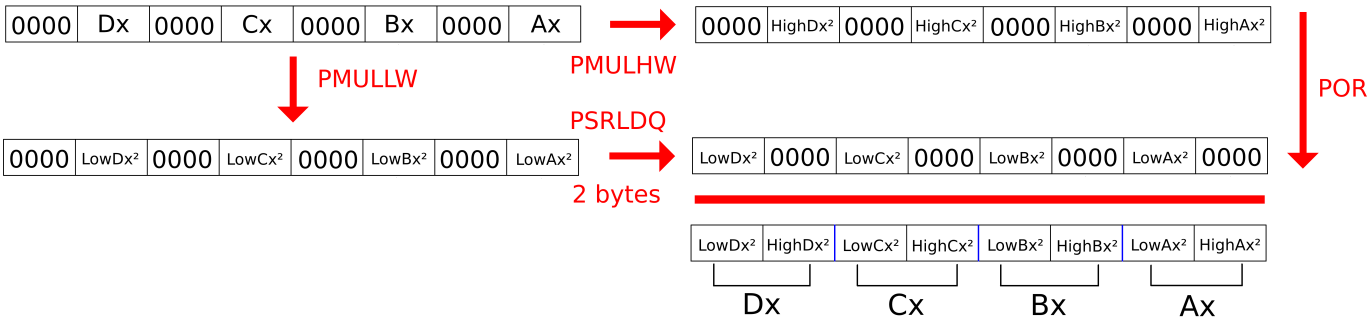


Figura 5: Cálculo del cuadrado para la parte baja de G_x. La parte alta es análoga y el caso para G_y es análogo al de G_x.

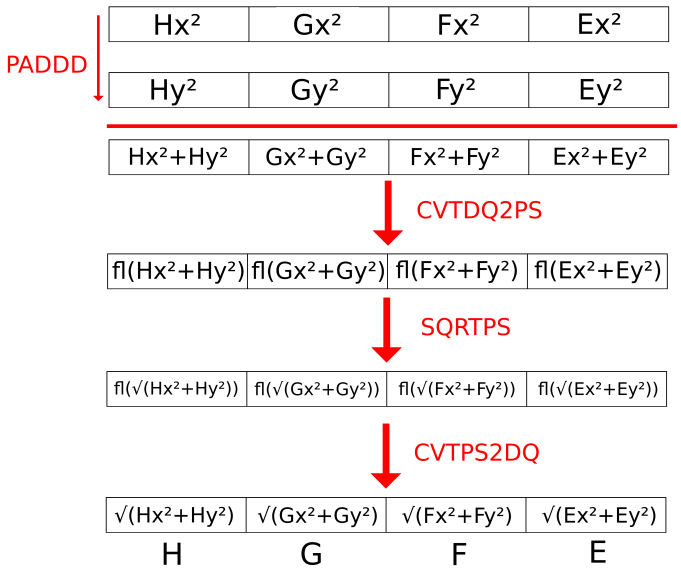
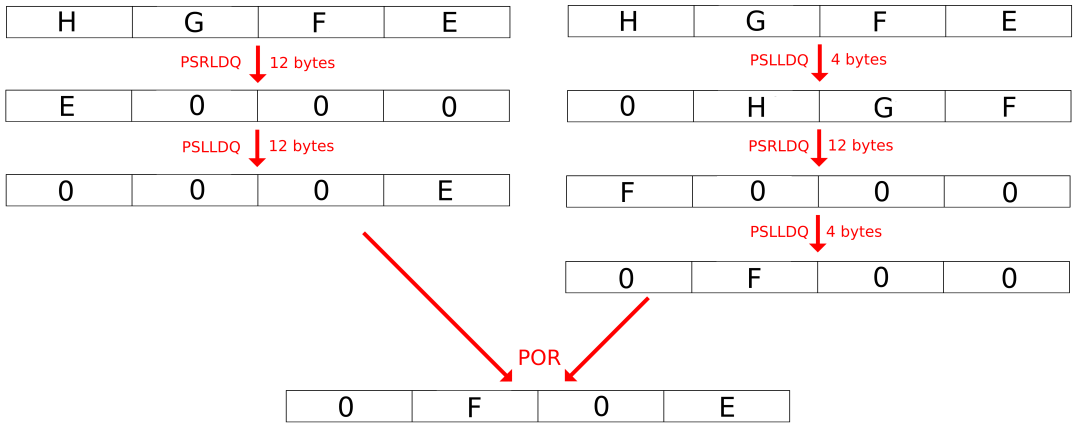
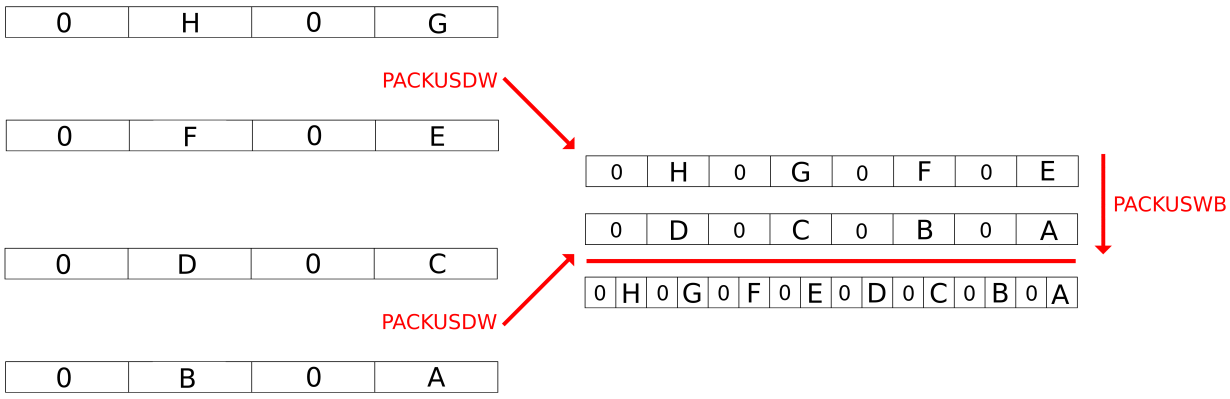


Figura 6: Cálculo de la raiz cuadrada de la suma de G_x y G_y con paso por punto flotante. El caso de A, B, C y D es análogo.



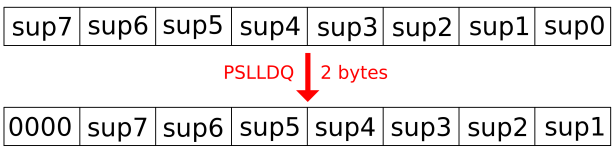
(a) Posicionamiento de los resultados a través de shifts para F y E. El caso de H y G es análogo y se repite lo mismo para D, C, B y A.



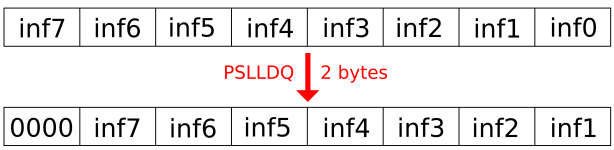
(b) Empaquetamiento de los resultados A, B, C, D, E, F, G y H en las posiciones correctas.

Figura 7: Ubicación de resultados para píxeles pares.

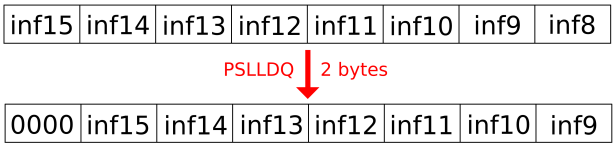
Hasta aquí se han calculado las posiciones pares (Figura 1a) pero resta obtener las impares (Figura 1b). Para ello se aplica un shift a cada uno de los registros con los píxeles de la imagen source (xmm1, xmm2, xmm3 y xmm4) como se muestra en la Figura 8. Luego de esto se trabaja de la misma manera que se explicó anteriormente para las posiciones pares, obteniéndose por último siete píxeles procesados correspondientes a cada letra de la Figura 1b y la última posición con una valor igual a cero pues inicialmente en los lugares que generan ese resultado había ceros. En última instancia se aplica un shift a estos siete resultados para que queden intercalados respecto del resultado para las posiciones pares y se aplica la operación POR a fin de obtener los 16 bytes que se imprimirán en la imagen destino.



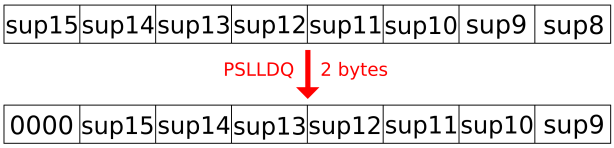
(a) Shift para la parte baja de la fila superior.



(b) Shift para la parte baja de la fila inferior.



(c) Shift para la parte alta de la fila inferior.



(d) Shift para la parte alta de la fila superior.

Figura 8: Desplazamiento sobre los valores de los píxeles de la imagen fuente.

2.2. Operadores Sobel y Prewitt

Los operadores Sobel y Prewitt son métodos utilizados para detección de bordes. Aplicándolos sobre una imagen digital en escala de grises, calculan el gradiente de intensidad de brillo de cada pixel. Este valor lo obtienen discretizando las derivadas de la imagen a través de la apliación de dos matrices de convolución de 3x3 (una para los contornos horizontales y otra para los verticales). El resultado muestra que tan abruptamente cambia una imagen en cada punto analizado.

$$\text{Sobel:} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \text{y} \quad G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

$$\text{Prewitt:} \quad G_y = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad \text{y} \quad G_x = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix}$$

Como vemos, usan casi las mismas matrices de convolución. Por lo tanto de aquí en adelante hablaremos solo de uno de ellos, pero sepamos que los todos los métodos de resolución son ídenticos. Lo único que se modifica es la matriz de convolución a utilizar.

La idea del operador es la siguiente:
Para cada píxel (i, j) de la imagen I, se toma la vecindad de 3x3 utilizando dicho píxel como central, y se lo multiplica por las matrices de convolución:

$$I(i,j) = \begin{matrix} \vdots & \vdots & \vdots & \\ \cdots & I_{i-1,j-1} & I_{i-1,j} & I_{i-1,j+1} & \cdots \\ \cdots & I_{i,j-1} & I_{i,j} & I_{i,j+1} & \cdots \\ \cdots & I_{i+1,j-1} & I_{i+1,j} & I_{i+1,j+1} & \cdots \\ \vdots & \vdots & \vdots & \end{matrix}$$

llamemos, $G_x * I(i,j) = G_x(i,j)$ y $G_y * I(i,j) = G_y(i,j)$

Una vez obtenido $G_x(i,j)$ para el píxel, se procede a encontrar el módulo del gradiente calculado discretamente en el píxel (i,j) del siguiente modo:
$Op(i,j) = \sqrt{G_x(i,j)^2 + G_y(i,j)^2}$
Finalmente, btenemos el valor del píxel (i,j) de la imágen luego de aplicarle el operador. Notemos que al tener que disponer de una vecindad de píxeles de 3x3, entonces debemos evitar los píxeles del borde (ya sea horizontal o vertical).

2.2.1. Algoritmo en C

Eh aquí el pseudo-código del operador Sobel:

Algoritmo 2 Sobel en C++

Aclaraciones: $*src$ es un puntero a la imagen fuente, $*dst$ es un puntero a la imagen destino, m y n son alto y ancho de la imagen en cantidad de píxeles respectivamente, row_size hace referencia (tanto para src como para dst) al ancho de la imagen incluyendo el posible padding.

```
1:  procedure SOBEL C(*src, *dst, m, n, src_row_size, dst_row_size)
2:      ancho = 0
3:      alto = 0
4:      rellenarConCeros(primerFila)
5:      rellenarConCeros(primerColumna)
6:      rellenarConCeros(ultimaFila)
7:      rellenarConCeros(ultimaColumna) ▷ Los ciclos llegan hasta las anteúltimas fila y columna pues éstas
    no se procesan
8:      while alto <  m − 1 do
9:          while ancho < n − 1 do
10:             g_x = − ∗ (src − src_row_size − 1) + ∗(src − src_row_size + 1)
11:                   −2 ∗ (∗(src − 1))            + 2 ∗ (∗(src + 1))
12:                   − ∗ (src + src_row_size − 1) + ∗(src + src_row_size + 1)
13:
14:             g_y = ∗(src − src_row_size − 1) + 2 ∗ (∗(src − src_row_size))
15:                   + ∗(src − src_row_size + 1) − ∗(src + src_row_size − 1)
16:                   − 2 ∗ (∗(src + src_row_size)) − ∗(src + src_row_size + 1)
17:
18:             *dst = sqrt((g_x)^2 + (g_y)^2)
19:             src = src + 1
20:             dst = dst + 1
21:             ancho = ancho + 1
22:         src   += src_row_size
23:         dst   += dst_row_size
24:         alto   += 1
25: end procedure
```

2.2.2. Algoritmo en Assembly

En esta sección detallaremos los pasos realizados para el cálculo del operador en Assembly. Explotaremos la utilización de instrucciones SIMD con el fin de obtener una optimización lo mayor posible.

Como vimos anteriormente, para procesar cada píxel se necesita un kernel de 3x3 donde éste sea el píxel central. Esto nos lleva a concluir que en cada acceso a memoria que se haga, tendremos que levantantar no solo los 16 bytes a procesar, sino que sus vecinos de arriba y de abajo. También hay que tener en cuenta que el procesamiento del píxel utiliza los vecinos inmediatos de la izquierda y derecha. Es por esto último que podemos decir que, levantando los 16 bytes de los píxels a procesar, los 16 bytes de arriba y los 16 bytes de abajo, podremos procesar solamente los 14 bytes de los píxeles a procesar. Aquí un dibujo en donde se muestra lo recientemente propuesto de una forma más descriptiva:

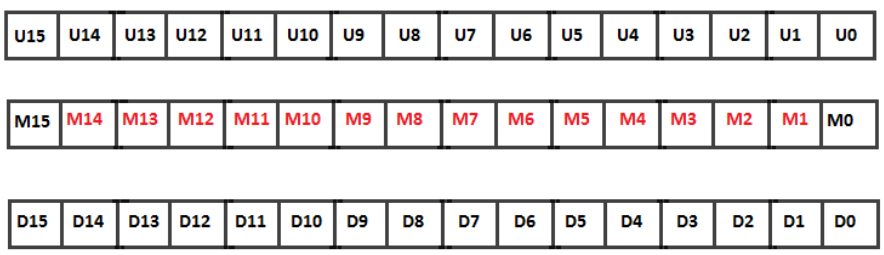


Figura 9: En rojo, los píxeles a los cuales se les aplicará la convolución

De aquí en adelante llamaremos:

- U_i con $i = 0.,15$ a los píxeles vecinos de arriba.
- M_i con $i = 0.,15$ a los píxeles a procesar.
- D_i con $i = 0.,15$ a los píxeles vecinos de abajo.

Para un más claro el entendimiento del algoritmo lo dividiremos en 3 partes:

- Cálculo de $G_x(i, j)$
- Cálculo de $G_y(i, j)$
- Unión y Parte Final

Antes de seguir, lo primero que haremos será extender la precisión de byte a word, ya que al tener que hacer operaciones podríamos estar perdiendo información debido a la corta presición.

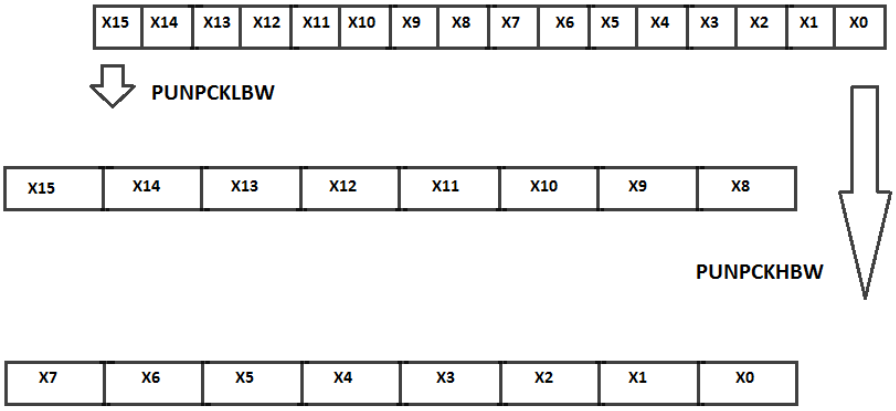


Figura 10: Esto se debe realizar con $X = D$, $X = U$ y $X = M$

Cálculo de $G_x(i, j)$

La ecuación correspondiente a la convolución definida por $G_x(M_i)$ para cada píxel a procesar, es la siguiente:

$$G_x(M_i) = U_{i-1} + 2 * M_{i-1} + D_{i-1} - (U_{i+1} + 2 * M_{i+1} + D_{i+1})$$

por lo tanto, el objetivo será poder calcular dicha expresión para los píxeles M_i con $i = 1 \dots 14$

En el punto uno de la figura 11, se calcula el doble del valor del píxel a procesar. En el punto dos hacemos la suma de los tres registros, acercandonos al valor final de G_x según la ecuación más arriba nombrada. En el punto tres, ejecutamos un corrimiento, con el fin de acomodar los indices. Según la ecuación de G_x para un píxel de índice i se necesita el valor de la suma del píxel U,D, y M de los índices $_{i-1}$ e $_{i+1}$. Por lo tanto, efectuando un corrimiento de dos words hacia la derecha, logramos alinear los índices de la forma deseada.

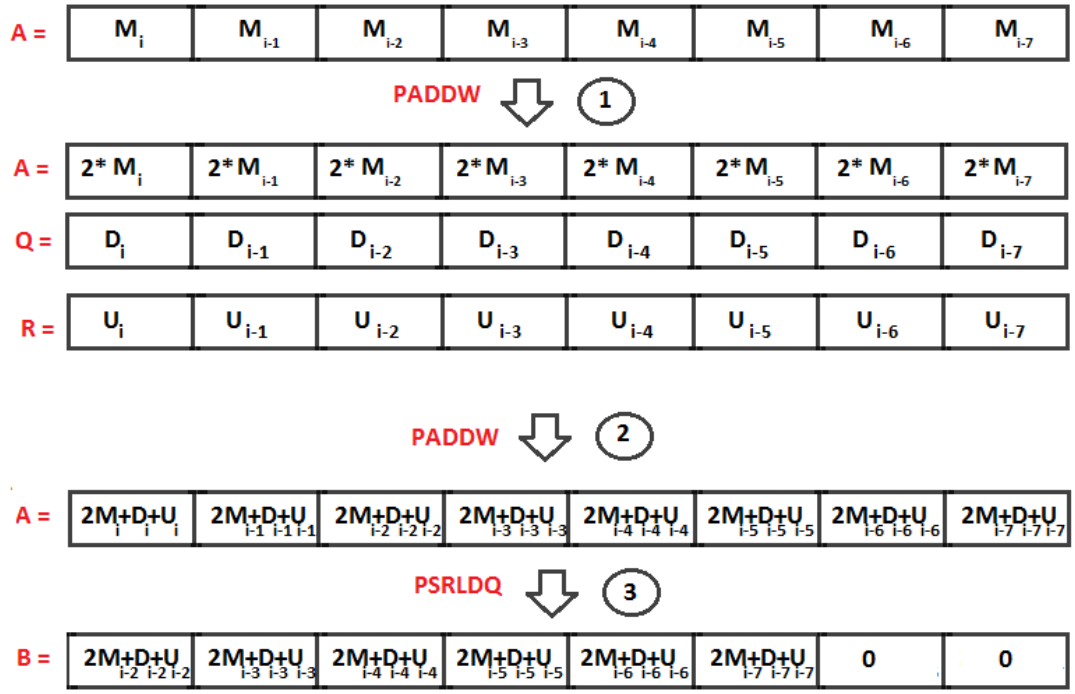


Figura 11: Esto se realiza tanto con $i = 15$ como con $i = 7$

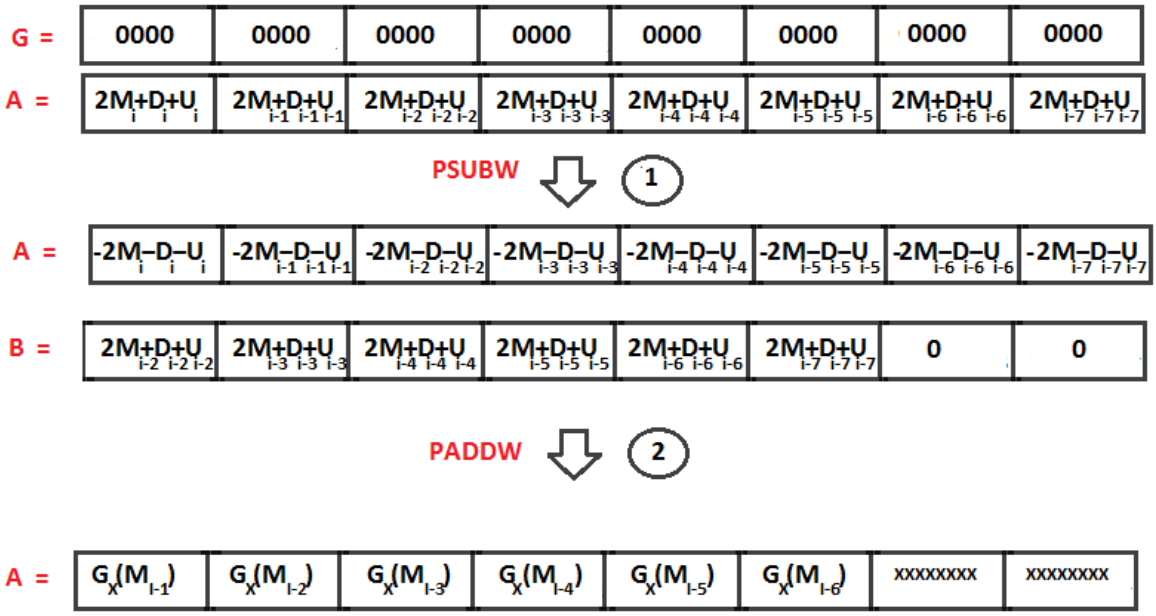


Figura 12: Esto se realiza tanto con $i = 15$ como con $i = 7$

Notemos que con la suma antes calculada, tenemos toda la información para poder armar la ecuación de G_x. En el registro A guardamos la suma calculada en el punto anterior antes de hacer el corrimiento. En el punto uno obtenemos el inverso aditivo de dicho valor. El punto dos es el que representa la suma de el registro A con el registro obtenido en el punto anterior. El resultado que se obtiene es la ecuación de G_x para los M_i con $i = 14, 13, 12, 11, 10, 9, 6, 5, 4, 3, 2, 1$. Recordemos que los píxeles $M_1 5$ y M_0 no son calculados ya no tenemos información sobre sus vecinos. Pero, ¿Qué pasó con los píxeles M_8 y M_7?
El problema fué que al realizar la extensión de byte a word, se dividió un solo registro (donde teníamos los valores iniciales obtenidos de memoria) en dos nuevos registros. Como dijimos, los extremos no se pueden calcular ya que no se conoce información sobre sus vecinos. Al dividir un registro en dos, pasamos de tener dos extremos a cuatro. La diferencia es que de esos cuatro, dos son calculables puesto que sí poseemos información para hacerlo. Veamos cómo:

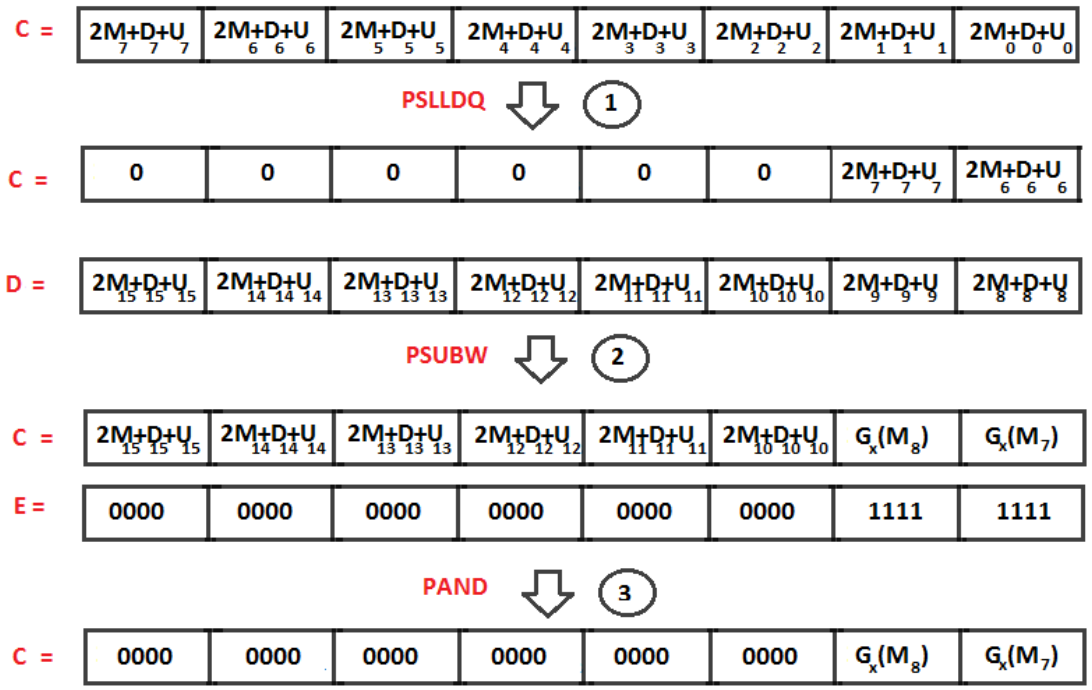


Figura 13: Cálculo de G_x para M_7 y M_8

Nuestro primer paso para el cálculo de M_8 y M_7 es usar un registro en el cual hallamos guardado el valor de la suma de $2*M_i+D_i+U_i$ con $i = 0.,7$ y aplicarle un corrimiento hacia la izquierda. De esta manera, obtendremos como muestra la figura 13, la suma para los índices $_7$ y $_8$ en la anteúltima y última word respectivamente.
Luego utilizando un nuevo registro el cual previamente se lo cargó con el valor de la suma de $2 * M_i + D_i + U_i$ con $i = 15.,8$ y notando que los índices quedan exactamente alineados como uno quiere, podemos realizar la operación de resta y calcular el valor de la ecuación G_x para los índices $_7$ y $_8$.
Por último, al registro resultante de la resta, se le aplica una máscara la cual mantiene el mismo valor en las últimas dos words (valores que nos interesan) y pone en 0 los demas bits. De esta forma ya tenemos los valores de G_x para los M_i con $i = 0 \dots 14$. Nos faltaria unir la información en solamente dos registros.

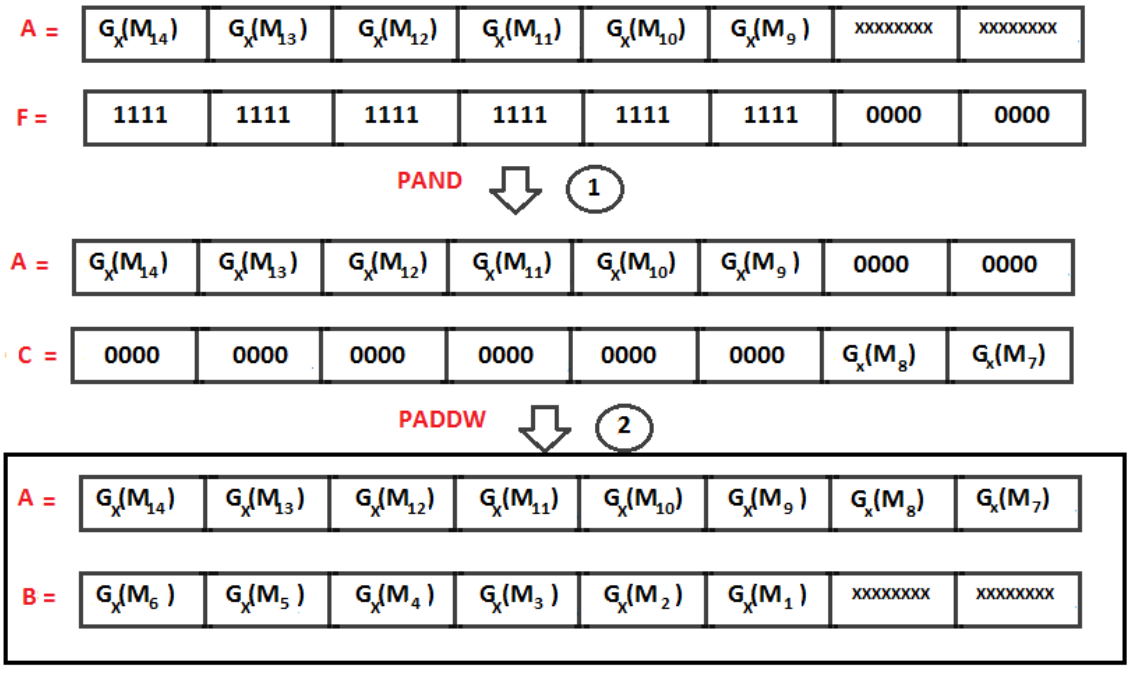


Figura 14: Unión y obtención de la totalidad de los G_x

Para terminar de unir la información aplicamos una máscara sobre el registro donde guardamos los valores $G_x(M_i)$ con $i = 9 \dots 14$ para mantener estos resultados y poner en 0 las últimas dos words.

Una vez aplicada la máscara podemos realizar la suma con el registro donde poseemos los valores de $G_x(M_7)$ y $G_x(M_8)$ sin problemas y obtener el resultado final en solo dos registros como indica la figura 14.

Cálculo de $G_y(i, j)$

La ecuación correspondiente a la convolución definida por $G_y(M_i)$ para cada píxel a procesar, es la siguiente:

$$G_y(M_i) = U_{i+1} + 2 * U_i + U_{i-1} - (D_{i+1} + 2 * D_i + D_{i-1})$$

por lo tanto, el objetivo será poder calcular dicha expresión para los píxeles M_i con $i = 1 \dots 14$

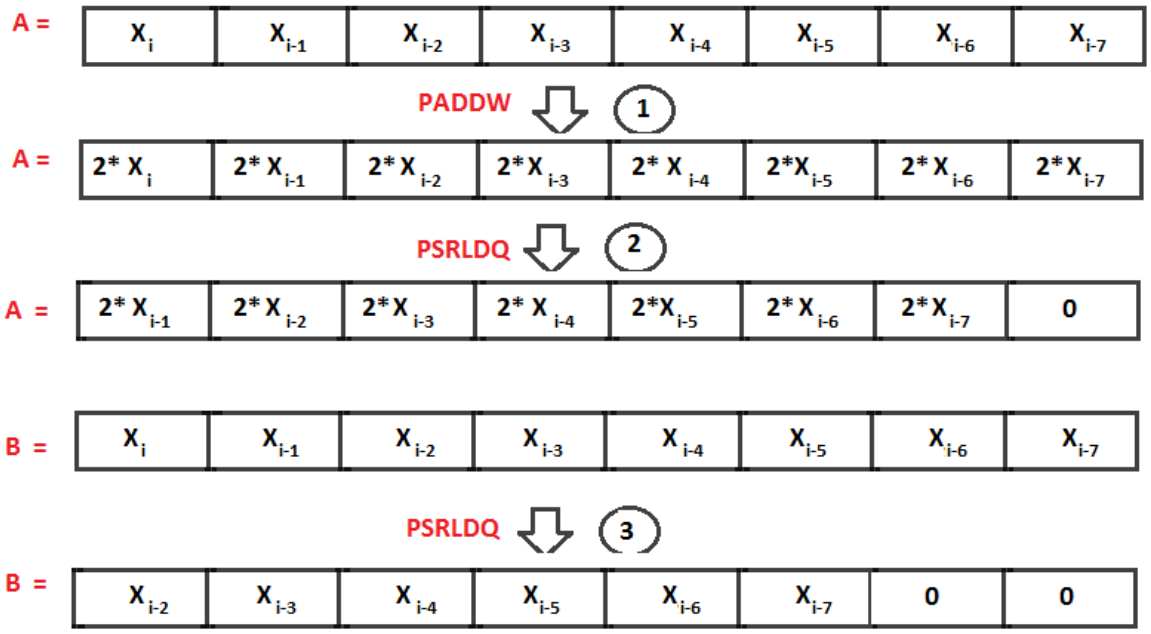


Figura 15: Esto se debe realizar con $X = D$, $X = U$ y $X = M$

En primera instancia a cada registro con los valores de los píxeles U_i y D_i los multiplicamos por dos y hacemos un corrimiento de un word hacia la derecha para acomodar los índices como la ecuación de G_y indica. En segunda instancia hacemos un corrimiento de dos words hacia la derecha a los registros que contienen los valores de los píxeles U_i y D_i.

Una vez echos estos cálculos ya tenemos la información necesaria para calcular la ecuación de G_y.

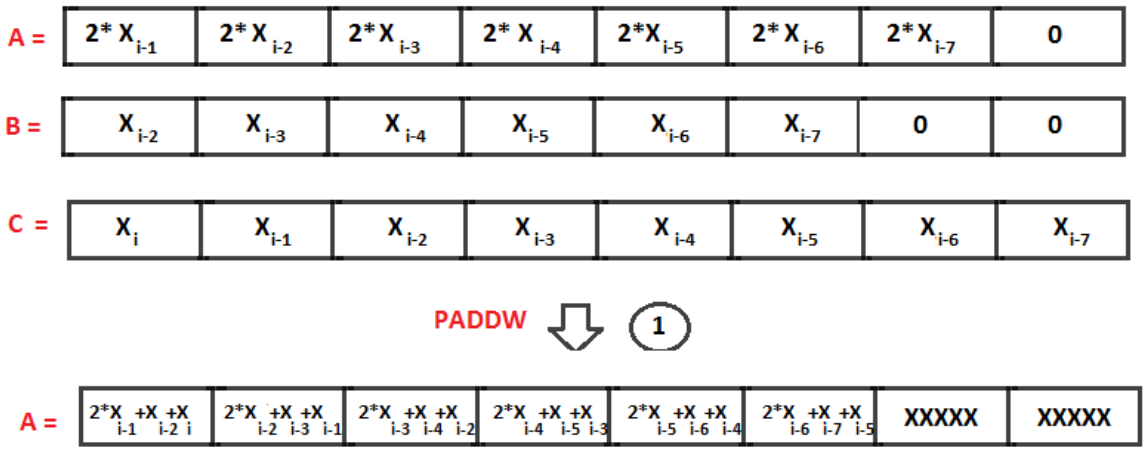


Figura 16: Esto se debe realizar con $X = D$, $X = U$ y $X = M$

Efectuamos la suma de los registros calculados y los registros con lo valores originales de U_i y D_i, obteneniendo una parte importante de la ecuación de G_y. Ya tenemos la información necesaria para calcular dicha ecuación de forma completa. Para ello, debemos sumar los valores obtenidos anteriormente de la siguiente manera:

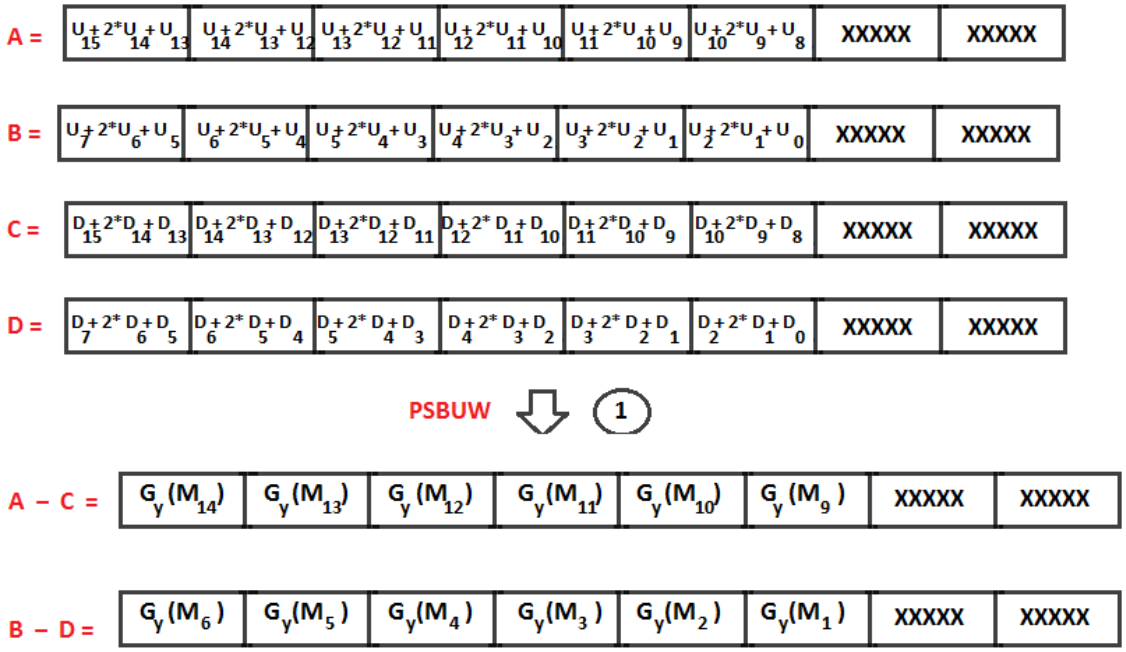


Figura 17: Cálculo de los primeros G_y

De esta forma, al igual que en la sección anterior, obtenemos el resultado de la ecuación $G_y(M_i)$ para $i = 14, 13, 12, 11, 10, 9, 6, 5, 4, 3, 2, 1$.. La justificación de qué pasa con $i = 7$ e $i = 8$ es la misma que en al sección "Cálculo de G_y". La forma de calcular $G_y(M_7)$ y $G_y(M_8)$ difiere de la anterior y es levemente más compleja.

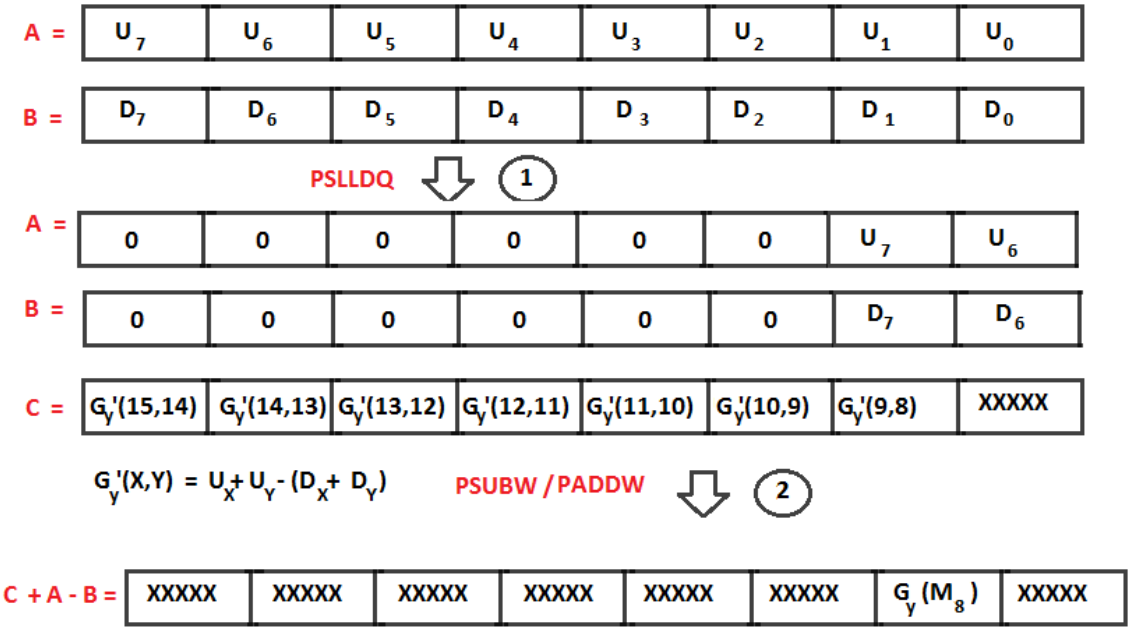


Figura 18: Cálculo de $G_y(M_8)$

El primer paso que damos para el cálculo de $G_y(M_8)$ es hacer un corrimiento de seis words hacia la izquierda sobre los registros que contienen U_i y D_i con $i = 0 \ldots 7$ a fin de acomodar los índices de la manera deseada. El siguiente paso es sumar los registros anteriormente nombrados con un tercer registro. Este registro "C" contiene el valor de un paso intermedio en el cálculo representado por la figura 16. El resultado de esta suma contiene el valor de $G_y(M_8)$ ubicado en la anteúltima word.
El cálculo de $G_y(M_7)$ no es tan trivial como el de $G_y(M_8)$. La estrategia a utilizar es similar a la que se usó para el calculo de $G_y(M_i)$ con $i = 14, 13, 12, 11, 10, 9, 6, 5, 4, 3, 2, 1$ pero utilizada en particular solo para $i = 7$.

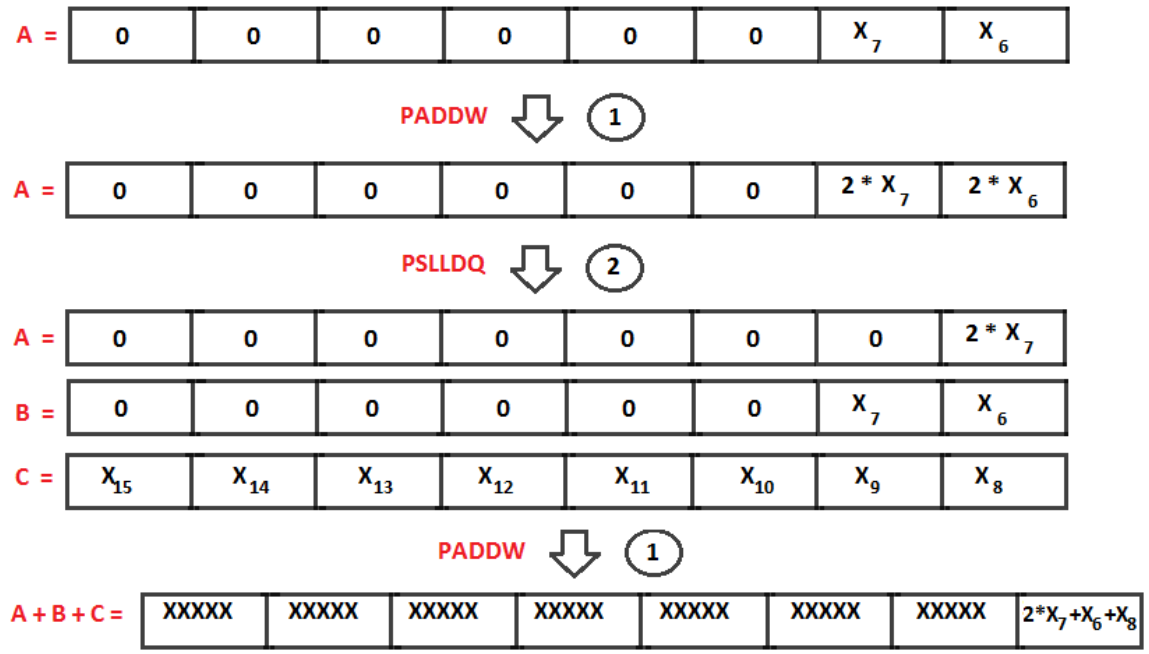


Figura 19: Cálculo de $G_y(M_7)$

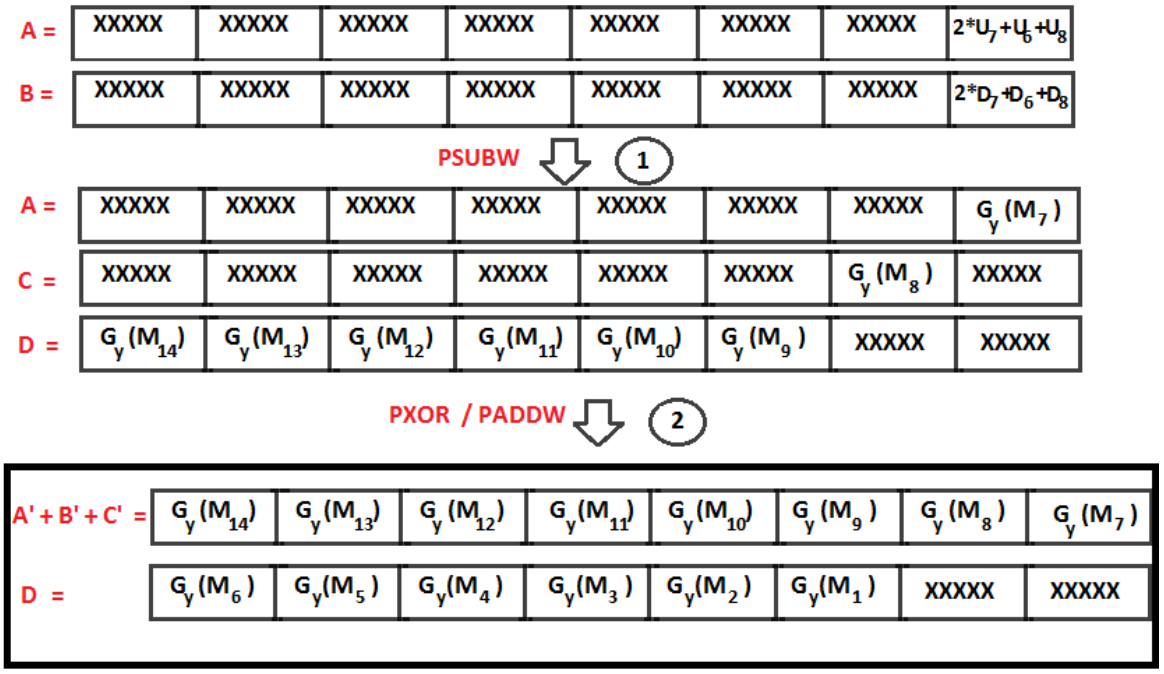


Figura 20: Unión y obtención de la totalidad de todos los G_y

Una vez que obtenemos la suma de los U_i y de los D_i con $i = 6, 7, 8$. Realizamos la operación resta y obtenemos como resultado el valor de $G_y(M_7)$.
A esta altura, tenemos todos los valores que necesitabamos calcular, falta unirlos. El cálculo de $G_y(M_7)$ se encuentra en un registro "A" que contiene valores que no nos interesan salvo en la ultima word donde se encuentra el valor del cual hablamos. Con el registro donde tenemos guardado $G_y(M_8)$ ocurre lo mismo, pero este valor está ubicado en la anteúltima word. Luego, el registro que contiene los valores de $G_y(M_i)$ con $i = 9 \dots 14$ "D" posee información que no nos interesa en las últimas dos words.
Tanto a "A" como a "C" y "D" se les aplica una máscara relativa a cada uno de ellos, con el fin de escribir ceros en los lugares que no nos interesan. Luego se procede a realizar la suma y obtener los dos registros que poseen los valores de $G_y(M_i)$ con $i = 0 \dots 14$

Unión y Parte Final

Ya tenemos la información para completar la ecuación que plantea el filtro Sobel, nos falta unirla para formarla:

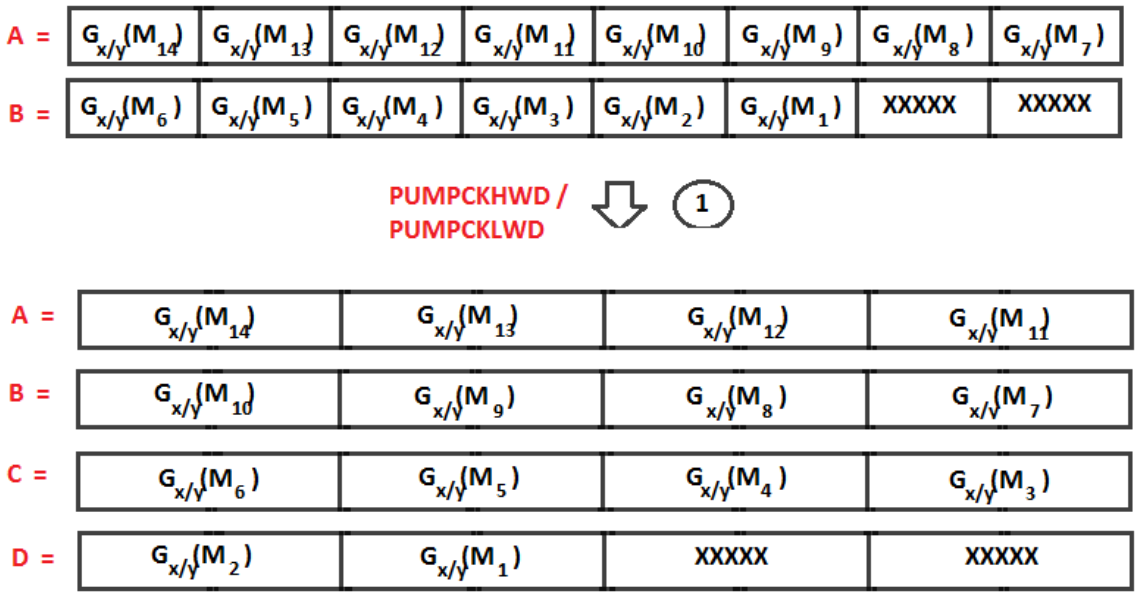


Figura 21: Extensión de word a double word.

Lo primero que hacemos es hacer "UMPCK" ya que trabajaremos con punto flotante, debido a que tendremos que aplicar raíz cuadrada según lo indica la ecuación planteada por el filtro Sobel. Es por eso que hacemos la extensión de words a double words.

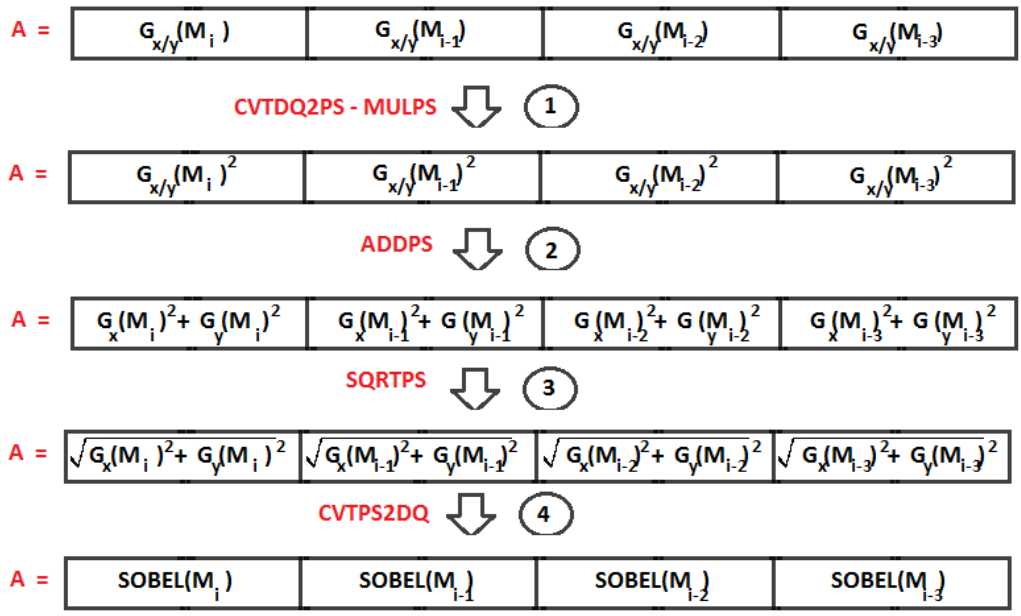


Figura 22: Utilizar $G_{x/y}(M_{i-3})$ en realidad es un abuso de notación ya que habrá registros que no exista dicho índice. El motivo de esta notación es mostrar la estructura de los registros.

En el primer paso, una vez echa la extensión, convertimos a punto flotante de presición simple y luego se multiplica el registro por si mismmo para elevarlo al cuadrado.

En el segundo, realizamos al suma de los registros que contienen los valores G_x con los G_y según su índice. Por ejemplo, un resultado que se obtendrá de esta suma será $G_x(M_8) + G_y(M_8)$, otro será $G_x(M_1) + G_y(M_1)$. El tercer paso es aplicarle raíz cuadrada a cada valor. Una vez realizado este paso, se puede decir que tenemos calculado el valor final de los 14 píxeles centrales. El cuarto y último paso, es volver a convertir los valores de punto flotante de presición simple a enteros de double words.

Por lo tanto, lo único que nos resta por hacer es volver a bytes con el fin de escribir en memoria:

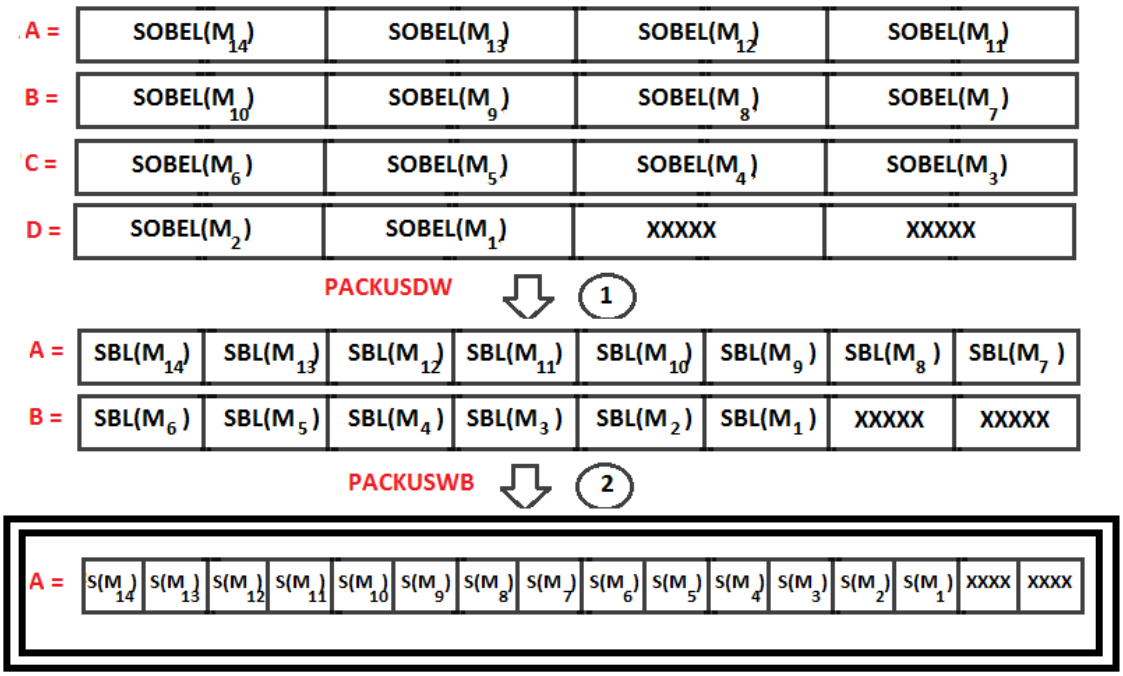


Figura 23: Compactación de double word a word y a byte de los resultados finales obtenidos.

Lo primero es volver de la extensión de double word a word. Para más tarde pasar de word a byte. Una vez que los valores sean convertidos en bytes, entonces estará listo para escribir en memoria.

2.3. Operador Canny

2.3.1. Explicación general de Canny

El filtro de imágenes propuesto por Canny se puede separar en cuatro partes:

1. Smoothing (Suavización)
2. Finding gradientes (Cálculo de los gradientes)
3. Non-maximum suppression with double thresholding (Supresión de píxeles "no máximos" con umbral doble)
4. Edge-tracking by histeresis (Seguimiento de bordes por histeresis)

Cada una de ellas cumple un rol fundamental en el procesamiento total. Pasaremos a detallar cada una a continuación.

Smoothing

Como su nombre lo indica, esta parte del filtro busca suavizar bordes. Esto se debe a que cualquier imagen tomada por una cámara tendrá cierto grado de ruido y de distintos tipos. Uno de ellos es el de tipo "sal y pimienta" (o "salt and pepper" en inglés) el cual se aprecia en la imagen como puntos negros en zonas claras y puntos blancos en zonas oscuras. Este tipo de ruido es provocado muchas veces por la conversión analógica-digital que se produce al momento de tomar la fotografía aunque también por bits de error durante la transmisión de los datos. Es por esto que, al trabajar con imágenes, es uno de los errores más comunes. Dado que se presenta como puntos que contrastan notablemente con la imagen, tenerlos en cuenta en cualquier filtro resultará en una propagación de errores. Sin embargo, estos errores se pueden evitar aplicando previamente un filtro de suavización.

El filtro utilizado en esta implementación fue el Gaussiano de 5x5 píxeles. Si bien la elección del tamaño influye en cuánta información tomar para suavizar la imagen a lo alto y a lo ancho, el tamaño de la máscara se ve fuertemente relacionado con el tamaño de la imagen. Valiéndonos de información recopilada en diversas fuentes escogimos la convolución de la Ecuación 1 debido a que consideramos un tamaño de 5x5 píxeles lo suficientemente grande como para eliminar el ruido de sal y pimienta en la mayoría de los casos y considerando además que para imágenes pequeñas el filtro no debe ser demasiado amplio a fin de no deformar la imagen.

$$S = \frac{1}{159}. \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \tag{1}$$

A continuación presentamos como ejemplo los resultados obtenidos al aplicar esta convolución sobre algunas imágenes.

(a) Imagen de lena 512x512 píxeles en blanco y negro.

(b) Imagen de lena 512x512 píxeles luego de suavizarla.

(c) Imagen de 170x227 píxeles en blanco y negro.

(d) Imagen de 170x227 píxeles luego de suavizarla.

Figura 24: Imágenes originales y suavizadas.

Finding gradients

La segunda parte del filtro consiste en calcular las direcciones de máximo crecimiento de la imagen. La idea de esto es poder hallar los lugares de la imagen donde la variación en la intensidad es mayor ya que esos lugares son bordes. Para esto se calcula el gradiente de la imagen de forma discreta puesto que éste representa los cambios en la imagen. El mismo está determinado por las dos convoluciones en Ecuación 2.

$$G_X = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \qquad G_Y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2}$$

Esos dos kernels son justamente los utilizados por el filtro de Sobel para hallar las componentes del gradiente horizontal y vertical respectivamente. Una vez calculadas, y por el Teorema de Pitágoras, se obtiene el ángulo del gradiente de la forma explicada en Ecuación 3.

$$\theta = \arctan\left(\frac{|G_Y|}{|G_X|}\right) \tag{3}$$

La tercer parte consiste en eliminar todo valor de la imagen que no se corresponda con un sector de la imagen de gradiente máximo. Para ello, en esta parte se discretiza el valor del ángulo del gradiente de la siguiente forma:

- Si $-22{,}5^o \leq \theta < 22{,}5$ ó $157{,}5 \leq \theta < 202{,}5$ entonces $valor \leftarrow 50$
- Si $22{,}5 \leq \theta < 67{,}5$ ó $202{,}5 \leq \theta < 247{,}5$ entonces $valor \leftarrow 100$
- Si $67{,}5 \leq \theta < 112{,}5$ ó $247{,}5 \leq \theta < 292{,}5$ entonces $valor \leftarrow 150$
- Si $112{,}5 \leq \theta < 157{,}5$ ó $292{,}5 \leq \theta < 337{,}5$ entonces $valor \leftarrow 200$

Así, se representa con el valor 50 a los píxeles con gradiente horizontal (–), con valor 100 a los píxeles con gradiente diagonal (/), con valor 150 a los píxeles con gradiente diagonal opuesto (\) y con valor 200 a los píxeles con gradiente vertical (|).

Luego, al finalizar esta parte se tienen dos imágenes: una con el módulo del gradiente en cada punto (según Ecuación 4) y otra con los valores según el ángulo de la dirección del gradiente.

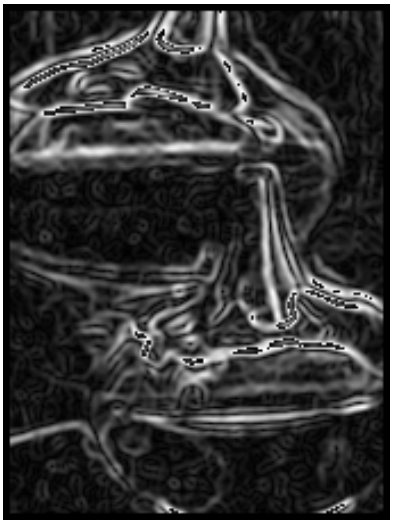
$$|G| = \sqrt{G_X^2 + G_Y^2} \tag{4}$$

A continuación se presentan las imágenes obtenidas al aplicar estas partes del algoritmo sobre las suavizadas anteriores.

(a) Imagen de lena 512x512 píxeles mostrando el valor del módulo del gradiente.

(b) Imagen de lena 512x512 píxeles mostrando el valor obtenido en función del ángulo del gradiente.

(c) Imagen de lena 512x512 píxeles mostrando el resultado al aplicar la umbralización.



(d) Imagen de 170x227 píxeles mostrando el valor del módulo del gradiente.

(e) Imagen de 170x227 píxeles mostrando el valor obtenido en función del ángulo del gradiente.

(f) Imagen de 170x227 píxeles mostrando el resultado al aplicar la umbralización.

Figura 25: Gradiente, ángulo y umbralización de las imágenes.

Non-maximum suppression with double thresholding

En la tercer etapa se pasa a determinar aquellos píxeles que son borde respecto de sus vecinos. Eso se logra comparando el valor del módulo del gradiente con dos vecinos de acuerdo a lo siguiente:

- Si $valor = 50$, se compara con los píxeles a izquierda y derecha (4 y 6 en Figura 26)
- Si $valor = 100$, se compara con los píxeles a izquierda-abajo y derecha-arriba (7 y 3 en Figura 26)
- Si $valor = 150$, se compara con los píxeles a arriba y abajo (2 y 8 en Figura 26)
- Si $valor = 200$, se compara con los píxeles a arriba-izquierda y abajo-derecha (1 y 9 en Figura 26)

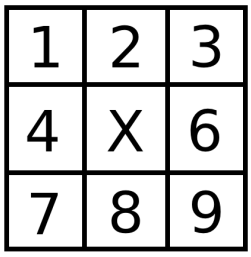


Figura 26: Posiciones relativas al píxel con el que se procesa. La "X" representa a dicho píxel.

Si la comparación indica que el píxel es mayor a sus dos vecinos, entonces eso es porque respecto de ese crecimiento (dirección del gradiente) el píxel es máximo local y en consecuencia un punto de cambio brusco en la imagen.

En esta parte del algoritmo también entran en juego dos valores más: el umbral inferior (THRESHOLD_LOW) y el umbral superior (THRESHOLD_HIGH). Para aquellos píxeles cuyo módulo del gradiente supera el umbral superior se determina el valor 200, para aquellos menores a dicho valor pero superiores al umbral inferior se determina el valor 100 y para el resto se determina el valor 0.

El objetivo de esta umbralización es poder evitar bordes falsos obtenidos gracias a ruido en la imagen, variaciones del color o incluso superficies poco regulares que si bien resultan en bordes tal vez no sean de interés para la aplicación del filtro. Así, la selección de los valores que determinan la ventana de umbral permiten ajustar los valores que deben tener los bordes que se busca observar. Vale aclarar que ventanas muy grandes no resultan en beneficio pues prácticamente cualquier variación es tomada como un borde perdiendo definición en la detección de bordes aunque, por otro lado, ventanas muy pequeñas tal vez no permitan observar lo necesario.

Una vez pasada esta etapa se tiene una única imagen con tres posibles valores: nulo, bordes débiles y bordes fuertes. A continuación se presentan imágenes con

Edge-tracking by histeresis

La última parte del algoritmo consiste en realizar una selección de bordes débiles para transformarlos en bordes fuertes. La idea es que un píxel de borde débil se transforma en borde fuerte si y sólo si está conectado a un borde fuerte. El criterio detrás de esta elección es que si bien hay bordes débiles que se deben a ruido y otras pequeñas variaciones, si éstos se encuentran cercanos a bordes fuertes entonces con gran probabilidad sean bordes de interés pero que no tuvieron la intensidad necesaria (respecto de su entorno) para ser considerados bordes fuertes.

A fin de no considerar píxeles muy lejanos y, en consecuencia, marcar como píxeles fuertes a muchos que no deberían serlo, nos limitamos a tomar una vecindad de 1 píxel para cada uno. Dicha vecindad es análoga a la presentada en la Figura 26.

A continuación se presentan ejemplos del procesamiento en este caso para las imágenes antes observadas.

(a) Imagen de lena 512x512 píxeles una vez umbralizada.

(b) Imagen de lena 512x512 píxeles luego de aplicar el edge-tracking.

(c) Imagen de 170x227 píxeles una vez umbralizada.

(d) Imagen de 170x227 píxeles luego de aplicar el edge-tracking.

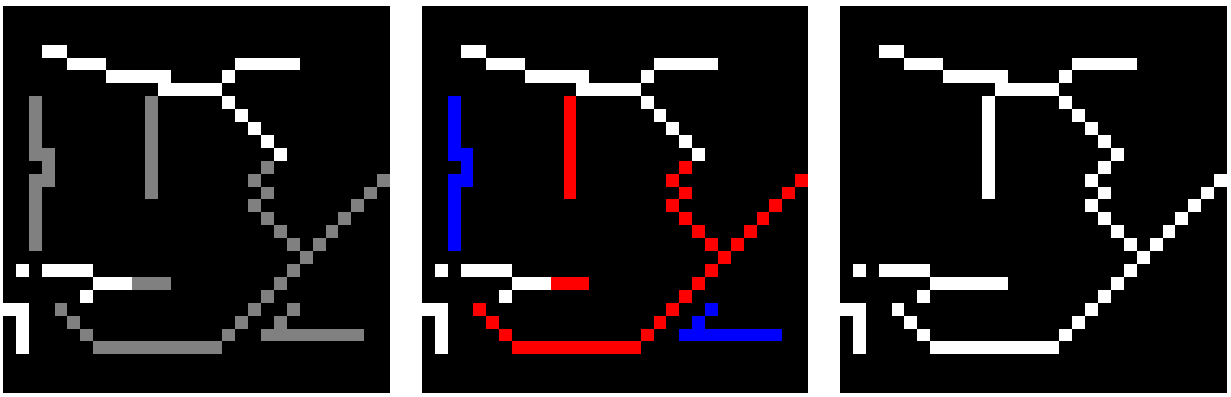
Figura 27: Imágenes umbralizadas y finales. En todos los casos THRESHOLD_HIGH es 120 y THRESHOLD_LOW es 55

2.3.2. Implementación en general

De las cuatro partes del algoritmo, la primera y la tercera se implementan tanto en C como en Assembly. La segunda etapa es idéntica a la ya explicada en la sección del algoritmo se Sobel con la única diferencia de que ahora se calcula también el ángulo. Mostraremos (tanto en C como en Assembly) el código que se agrega para realizar este cálculo. Por último, la cuarta etapa se presenta programada únicamente en C. La razón detrás de esta decisión está relacionada con el procedimiento que se debe realizar en esa parte en el cual no se puede aprovechar la tecnología SIMD pues se debe trabajar de manera independiente con cada píxel.

Habiendo dicho eso, pasamos a explicar el funcionamiento del algoritmo en su cuarta parte ahora y luego en cada una de sus implementaciones para el resto de las etapas.

El funcionamiento de doubleThresholding consiste en darle valor de borde fuerte a todos los píxeles marcados como de borde débil que se encuentren en contacto con algún borde fuerte. Sin embargo, esto no debe limitarse solamente a la vecindad estricta sino que cualquier camino débil que tenga a algún vecino que forme parte de un borde fuerte debe pasar a ser fuerte. Un ejemplo de esto puede observarse en la Figura 28.



(a) Imagen con bordes débiles (gris) y bordes fuertes (blanco).

(b) Misma imagen donde los bordes débiles vecinos a bordes fuertes se marcaron con rojo y los débiles no vecinos a fuertes se marcaron con azul.

(c) Misma imagen con el resultado final donde los bordes débiles que eran vecinos a bordes fuertes ahora son fuertes y los demás fueron eliminados.

Figura 28: Proceso de diferenciación entre bordes débiles vecinos a bordes fuertes y bordes débiles no vecinos a bordes fuertes.

La idea de algoritmo entonces es recorrer la imagen con la técnica de Depth First Search (DFS) [4] si uno considera a la imagen como un grafo planar donde cada vértice representa a un píxel y dos vértices son adyacentes si y sólo si los píxeles que representan son vecinos. Diremos que los vértices A e B son vecinos cuando A ocupa la posición X y B cualquiera de las otras posiciones en la Figura 26 o viceversa.

A fin de marcar a los píxeles ya recorridos se utiliza una matriz booleana del mismo tamaño que la imagen donde el valor *true* indica que el píxel debe ser recorrido y *false* cuando esto ya fue hecho. Inicialmente los bordes se inician como ya recorridos pues allí se sabe que no habrá bordes (debido a que los filtros anteriores dejan esos valores como 0) y el resto de la imagen se inicia como aún no recorrida.

La idea entonces es recorrer toda la imagen y para cada píxel fuerte (valor 200) se recorren sus vecinos y se "pushean" a la pila aquellos que tienen el valor 100, es decir, aquellos que son un borde débil. Una vez hecho eso para el píxel (i, j) se opera de la misma manera con las posiciones de píxeles en la pila (hasta que ésta se vacía) a fin de poder recorrer todos los caminos de bordes débiles con los que se tiene algún contacto e ir marcando cada píxel como borde fuerte. Como este procedimiento se repite para cada píxel aún no recorrido y que además tiene valor 200 entonces todos los píxeles de la imagen serán tenidos en cuenta y por ende si hay algún contacto con algún borde fuerte por parte de alguno débil entonces esto será notado o bien en la recorrida inicial o bien a través del recorrido por la pila. Vale aclarar que cualquier píxel que tenga el valor 0 al momento de recorrer la matriz será marcado como recorrido pues no podrá cambiar su valor ni hacer que otros lo cambien.

Por último, una vez operado con todos los posibles píxeles de la imagen es posible que haya algunos que hayan quedado con valor 100. Estos píxeles son aquellos débiles que no están en contacto con ningún fuerte y por ende hay que marcarlos como negro. Para hacer esto, se recorre toda la imagen y se da el valor 0 a todo píxel de valor 100.

A continuación se observa el pseudocódigo de la función doubleThresholding en la que se puede observar todo lo antes descrito.

Algoritmo 3 doubleThresholding

Aclaraciones: $*src$ es un puntero a la imagen fuente, $*dst$ es un puntero a la imagen destino, $cantFilas$ y $cantColumnas$ son alto y ancho de la imagen en cantidad de píxeles respectivamente, src_row_size hace referencia (tanto para src omo para dst) al ancho de la imagen incluyendo el posible padding.

```
1:  procedure SMOOTHING C(*src, cantFilas, cantColumnas, src_row_size)
2:      itColumnas = 0
3:      itFilas = 0
4:      itSrc = src
5:      porRevisar[cantFilas][cantColumnas] inicializado con false en los bordes y true el resto
6:
7:      for i = 1; i < cantFilas − 1 do
8:          for j = 1; j < cantColumnas − 1 do
9:              pos = src + i ∗ src_row_size + j
10:             if porRevisar[i][j] ∧ ∗pos = 0 then
11:                 porRevisar[i][j] = false
12:             else if porRevisar[i][j] ∧ ∗pos = 200 then
13:                 Para cada vecino del píxel (i, j) que tenga valor 100 apilar su posición como (abscisa, ordenada)
14:                 porRevisar[i][j] = false
15:                 while !laPila.EsVacia() do
16:                     abc = laPila.AbcisaTope()
17:                     ord = laPila.OrdenadaTope()
18:                     pos = src + abc ∗ src_row_size + ord
19:                     laPila.SacarTope()
20:                     if porRevisar[abc][ord] then
21:                         ∗pos = 200
22:                         Para cada vecino del píxel (abc, ord) que tenga valor 100 apilar su posición
23:                         porRevisar[abc][ord] = false
24:     for i = 1; i < cantFilas − 1 do
25:         for j = 1; j < cantColumnas − 1 do
26:             if ∗(src + i ∗ src_row_size + j) = 100 then
27:                 ∗(src + i ∗ src_row_size + j) = 0
28: end procedure
```

2.3.3. Algoritmos en C

El algoritmo resulta bastante simple pues expresa exactamente lo explicado al comienzo de la subsección 2.3.1 por lo que no ahondaremos en detalles. Los siguientes pseudocódigos explicitan la forma en la que se opera en cada una de las etapas.

Algoritmo 4 smoothing en C

Aclaraciones: $*src$ es un puntero a la imagen fuente, $*dst$ es un puntero a la imagen destino, $cantFilas$ y $cantColumnas$ son alto y ancho de la imagen en cantidad de píxeles respectivamente, src_row_size hace referencia (tanto para src omo para dst) al ancho de la imagen incluyendo el posible padding.

```
1: procedure SMOOTHING C(*src, *dst, m, n, src_row_size, dst_row_size)
2:     itColumnas = 0
3:     itFilas = 0
4:     itDst = dst
5:     itColumnas = 0
6:
7:     rellenarConCeros(primerFila)
8:     rellenarConCeros(segundaFila)
9:     rellenarConCeros(anteúltimaFila)
10:    rellenarConCeros(últimaFila)
11:
12:    dst = dst + 2 * dst_row_size
13:    src = src + 2 * src_row_size
14:    itDst = dst
15:
16:    itFilas = 2
17:    while itFilas < cantFilas − 2 do
18:        itColumnas = 0
19:        itSrc = src
20:        itDst = dst
21:        while itColumnas < cantColumnas do
22:            mascara = 0
23:            if itColumnas no es 0 ni 1 ni cantColumnas − 2 ni cantColumnas − 1 then
24:                mascara = pos1 * 2 + pos2 * 4 + pos3 * 5 + pos4 * 4 + pos5 * 2+
25:                          pos6 * 4 + pos7 * 9 + pos8 * 12 + pos9 * 9 + pos10 * 4+
26:                          pos11 * 5 + pos12 * 12 + pos13 * 15 + pos14 * 12 + pos15 * 5+
27:                          pos16 * 4 + pos17 * 9 + pos18 * 12 + pos19 * 9 + pos20 * 4+
28:                          pos21 * 2 + pos22 * 4 + pos23 * 5 + pos24 * 4 + pos25 * 2
29:            *itDst = (mascara/159) + 0,5
30:            itSrc + +
31:            itDst + +
32:            itColumnas + +
33:        src = src + src_row_size
34:        dst = dst + dst_row_size
35:        itFilas + +
36: end procedure
```

Donde las posiciones son las determinadas por la siguiente matriz: $\begin{bmatrix} pos1 & pos2 & pos3 & pos4 & pos5 \\ pos6 & pos7 & pos8 & pos9 & pos10 \\ pos11 & pos12 & pos13 & pos14 & pos15 \\ pos16 & pos17 & pos18 & pos19 & pos20 \\ pos21 & pos22 & pos23 & pos24 & pos25 \end{bmatrix}$

Algoritmo 5 gradienteYÁngulo en C

Aclaraciones: se presenta la única modificación en el código de Sobel. Ésta consiste en calcular el ángulo del gradiente justo antes de calcular el módulo del gradiente.

```
1: procedure GRADIENTEYÁNGULO C(*src, *grd, *ang, cantFilas, cantColumnas, src_row_size, grd_row_size, ang_row_size)
2:     angulo = atan2(G_y, G_x) * 180/PI        ▷ Dado que el resultado está en radianes lo transformo a grados
3:     if angulo < 0 then
4:         angulo = 360 + angulo                ▷ Si es negativo lo hace positivo (módulo 360°)
5:     if angulo = 255 then
6:         valor = 0                            ▷ Convención
7:     else if (0 ≤ angulo < 22,5) ∨ (157,5 ≤ angulo < 202,5) ∨ (337,5 ≤ angulo < 360) then
8:         valor = 50
9:     else if (22,5 ≤ angulo < 67,5) ∨ (202,5 ≤ angulo < 247,5) then
10:        valor = 100
11:    else if (67,5 ≤ angulo < 112,5) ∨ (247,5 ≤ angulo < 292,5) then
12:        valor = 150
13:    else if (112,5 ≤ angulo < 157,5) ∨ (292,5 ≤ angulo < 337,5) then
14:        valor = 200
15: end procedure
```

Algoritmo 6 nonMaxSup en C

Aclaraciones: $*grd$ es un puntero a la imagen con el módulo del gradiente, $*ang$ es un puntero a la imagen con el ángulo del gradiente, $*dst$ es un puntero a la imagen destino, $cantFilas$ y $cantColumnas$ son alto y ancho de la imagen en cantidad de píxeles respectivamente, grd_row_size hace referencia (tanto para grd omo para ang como para dst) al ancho de la imagen incluyendo el posible padding.

```
1: procedure nonMaxSup c(*grd, *ang, *dst, cantFilas, cantColumnas, grd_row_size, ang_row_size,
   dst_row_size)
2:     itColumnas = 0
3:     itFilas = 0
4:     rellenarConCeros(primerFila)
5:     rellenarConCeros(últimaFila)
6:     itDst = dst = dst + dst_row_size
7:     itGrd = grd = grd + grd_row_size
8:     itAng = ang = ang + ang_row_size
9:     itFilas = 1
10:    while itFilas < cantFilas − 1 do
11:        itColumnas = 0
12:        itGrd = grd
13:        itAng = ang
14:        itDst = dst
15:        while itColumnas < cantColumnas do
16:            valor = 0
17:            pixel = *itGrd
18:            if itColumnas no es 0 ni cantColumnas − 1 then
19:                if *itAng = 50 then
20:                    if *(itGrd − 1) < pixel > *(itGrd + 1) then
21:                        if pixel > THRESHOLD_HIGH then
22:                            valor = 200
23:                        else if pixel > THRESHOLD_LOW then
24:                            valor = 100
25:                    else if *itAng = 100 then
26:                        if *(itGrd − grd_row_size + 1) < pixel > *(itGrd + grd_row_size − 1) then
27:                            if pixel > THRESHOLD_HIGH then
28:                                valor = 200
29:                            else if pixel > THRESHOLD_LOW then
30:                                valor = 100
31:                            else if *itAng = 150 then
32:                                if *(itGrd − grd_row_size) < pixel > *(itGrd + grd_row_size) then
33:                                    if pixel > THRESHOLD_HIGH then
34:                                        valor = 200
35:                                    else if pixel > THRESHOLD_LOW then
36:                                        valor = 100
37:                            else if *itAng = 200 then
38:                                if *(itGrd−grd_row_size−1) < pixel > *(itGrd+grd_row_size+1) then
39:                                    if pixel > THRESHOLD_HIGH then
40:                                        valor = 200
41:                                    else if pixel > THRESHOLD_LOW then
42:                                        valor = 100
43:            *itDst = valor
44:            itGrd + +
45:            itAng + +
46:            itDst + +
47:            itColumnas + +
48:        grd = grd + grd_row_size
49:        ang = ang + ang_row_size
50:        dst = dst + dst_row_size
51:        itFilas + +
52: end procedure
```

2.3.4. Algoritmos en Assembly

Función Smoothing (smoothing_asm)

Para explicar la función `smoothing_asm` no incluiremos el código del recorrido de la imagen ya que la idea detrás del código es muy similar a la usada en otras funciones. La principal diferencia aquí tiene que ver con la cantidad de píxeles que se procesan. Dado que debe aplicarse la máscara de la Ecuación 1, y que esta tiene 5x5 píxeles de tamaño; entonces no se puede trabajar con las dos primeras filas y las dos últimas pues no tienen dos píxeles por encima y por debajo respectivamente. Dado que en esas zonas no se puede realizar el mismo procedimiento, entonces se decide darle a todos esos píxeles el valor 0. Análogamente se hace lo mismo para las dos primeras y las dos últimas columnas pues no tienen píxeles a izquierda y derecha respectivamente.

Por otro lado, a fin de poder aprovechar la tecnología SIMD se busca minimizar la cantidad de lecturas. Por el tamaño de la máscara, para procesar a cada píxel se requiere de las dos columnas a izquierda, de las dos a derecha, de las dos filas por encima y de las dos por debajo. En definitiva, para el procesamiento de un píxel se deben realizar lecturas sobre 5 filas diferentes. Sin embargo, leyendo de memoria 16 píxeles en 5 filas consecutivas se tiene información para procesar a los 12 píxeles centrales ya que sobre ellos es sobre los que se tiene leído su entorno de 2 píxeles en todas las direcciones. Esto se puede apreciar en la Figura 29. donde, por ejemplo, para procesar el píxel C8 se utilizan los píxeles: A6, A7, A8, A9, A10, B6, B7, B8, B9, B10, C6, C7, C8, C9, C10, D6, D7, D8, D9, D10, E6, E7, E8, E9, E10

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15

Figura 29: Posiciones en cinco filas consecutivas.

Si bien se pueden procesar los 12 píxeles centrales al hacer las 5 lecturas, esto es posible únicamente de a partes pues la utilización de las máscaras impide la utilización de los registros originales más de una vez. Es así que el procesamiento se realiza primero para las posiciones $C2$, $C7$ y $C12$; luego para las $C3$, $C8$ y $C13$; luego las $C4$ y $C9$; luego las $C5$ y $C10$ y por último para las $C6$ y $C11$. Esto se debe a que los procesamientos se deben separar en grupos de datos disjuntos y esta es una forma de hacerlo. Se puede observar en la Figura 30 cómo esto es resuelto.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15

(a) Posiciones en cinco filas consecutivas. Se marca entre líneas rojas a los píxeles necesarios para procesar a $C2$, $C7$ y $C12$.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15

(b) Posiciones en cinco filas consecutivas. Se marca entre líneas rojas a los píxeles necesarios para procesar a $C3$, $C8$ y $C13$.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15

(c) Posiciones en cinco filas consecutivas. Se marca entre líneas rojas a los píxeles necesarios para procesar a $C4$ y $C9$.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15

(d) Posiciones en cinco filas consecutivas. Se marca entre líneas rojas a los píxeles necesarios para procesar a $C5$ y $C10$.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	D13	D14	D15
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15

(e) Posiciones en cinco filas consecutivas. Se marca entre líneas rojas a los píxeles necesarios para procesar a $C6$ y $C11$.

Figura 30

Teniendo en cuenta esto, pasaremos a explicar el caso para $C2$, $C7$ y $C12$ ya que para los demás es análogo. No ahondaremos en detalles sobre el uso de los registros y las instrucciones porque son las mismas que para funciones anteriores por lo que sólo explicaremos la idea.

Dado que se debe multipicar a cada valor por los de la máscara (Figura 1), entonces se carga en registros los valores de cada una de las filas. Dado que se deben multiplicar entre sí, es necesario primero desempaquetar los registros de byte a double word (previo paso por word) y entonces recién ahí se opera. Como se tienen 16 bytes (correspondiente a 16 píxeles), es necesario repetir esa operación para cada grupo de 4 píxeles.

Debido a la cantidad de registros de los que se dispone no es posible realizar la multiplicación para cada parte en ese orden pues luego se deben sumar todos los resultados. Entonces se procede a multiplicar los 4 primeros con los correspondientes en las máscaras para todas las columnas y luego sumar todos esos resultados pudiendo guardar todo eso en un único registro. En la Figura 31 se observa cómo se hace eso con los primeros 4 aunque la operatoria es análoga para las otras tres tandas.

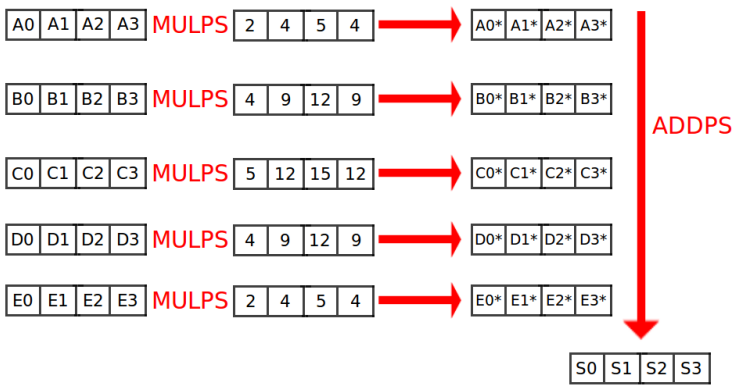


Figura 31: Modo de operación con la máscara para los primeros cuatro píxeles.

Una vez calculadas las cuatro tandas, se procede a sumar lateralmente para obtener para el píxel $C2$ la suma de las columnas 0, 1, 2, 3 y 4; para el píxel $C7$ la suma de las columnas 5, 6, 7, 8 y 9; para el píxel $C12$ la suma de las columnas 10, 11, 12, 13 y 14. Eso se resuelve usando shifts y sumas de la manera explicada en otras funciones.

Esto mismo se repite para los otros píxeles, por lo que al final se obtienen los datos para colocar en las posiciones $C2$ a $C13$. Como sólo son 12 píxeles los procesados, se colocan en la imagen destino y luego se avanzan los iteradores de las imágenes 12 posiciones.

Función Ángulo Sobel (anguloSobel_asm)

La función que detallaremos a continuación es la encargada de resolver la segunda parte según la enumeración mostrada al comienzo de la sección. Como ya hemos explicado anteriormente, esta función tomará una imagen de entrada (la obtenida por "Smoothing") y devolverá dos imágenes de salida (módulo del gradiente y valor según ángulo de dirección del gradiente). Por lo comentado algunos párrafos más arriba, en la sección "Finding Gradients" podemos dividir la función en tres pasos:

1. Cálculo del módulo del gradiente (operador Sobel).
2. Ecuación de Pitágoras.
3. Discretización del valor del ángulo.

Para poder hacer una explicación no repetitiva usaremos la implementación en Assembly del filtro Sobel, ya explicada en una sección anterior, y partiremos desde ese punto. Antes de comenzar, recordemos que procesaremos con 14 píxeles por cada vez que levantemos información de memoria.

Parte 1
Una vez calculados los valores de G_x y G_y para cada uno de los 14 píxeles, los almacenamos, calculamos el módulo del gradiente y más adelante recuperaramos los valores almacenados a fin de continuar con la segunda parte (valor del ángulo). Una vez obtenido el módulo del gradiente en cada punto, se procede a escribirlo en memoria. Hasta este punto, el código es el mismo que el descripto en la sección de Sobel (salvo por la parte de almacenar los valores G_x y G_y).

Parte 2
En primer lugar se recuperan los valores almacenados de G_x y G_y para los 14 píxeles. Teniendo en cuenta que a la hora de su almacenamiento estos datos se encontraban en formato de word, que pueden ser valores negativos y que más tarde trabajaremos con funciones que necesitarán formato de punto flotante (como arco tangente), se procede a hacer la extensión de word a double word respetando el signo y su posterior conversión a punto flotante de cada uno de los valores.

Dado que SSE no contiene una función para el cálculo del arco tangente, utilizamos la FPU. Ésta sólo puede operar con un valor a la vez, es decir que podemos trabajar de a un píxel. Es por esto, que para esta parte se procede a colocar los valores de G_x y G_y del mismo píxel en los resitros de la FPU, aplicar la función $FPATAN$ y almacenar su resultado. Es necesario remarcar que quizás no sea necesario el uso de esta función ya que:

- si G_x y G_y son 0 entonces el ángulo también lo será
- si solamente G_x es 0 entonces el ángulo será 90

Una vez calculados y almacenados los valores de los ángulos de los 14 píxeles se procede a multiplicar cada uno de estos valores por 180 y dividirlos por π a través del uso de máscaras. La razón por la cual debemos realizar este paso es que la función $FPATAN$ devuelve el valor del arco tangente en radianes, por lo tanto hacemos la conversión a grados (ya que con estos valores se opera en la tercera parte). Una vez realizada la conversión, es posible que alguno de estos valores sea negativo, si esto sucede, se procede a sumar 360 grados a los valores negativos con el fin de obtener el mismo ángulo pero con un valor positivo.

Parte 3

Una vez obtenidos los ángulos para cada píxel, buscamos a qué valor corresponden. Para ello se deberá encontrar el intervalo al cual el ángulo pertenece (según lo explicado en la parte de "Finding Gradients"). Todos los intervalos son disjuntos y además la union de ellos contempla todos los casos; por lo tanto, todos los ángulos pertenecerán a uno y sólo a uno de ellos. Además cada intervalo tendrá un solo valor asociado, sin embargo, varios intervalos pueden tener el mismo valor. Por ejemplo, si un ángulo está en el intervalo [67.5, 112.5) o en [247.5, 292.5) el valor será 150 para ambos.

Cómo se puede observar en la sección de "Finding Gradientes", para deducir a qué intervalo pertenece un ángulo de deben realizar dos comparaciones, una que pregunte por "mayor o igual" y otra por "menor estricto". Sabiendo que cada ángulo pertenece a un único intervalo, es que decidimos utilizar la siguiente estrategia:

Para cada registro R que contenga cuatro ángulos (o dos debido a que se procesan 14 píxeles ya que $14 \equiv 2(4)$), utilizaremos cuatro registros S_i donde cada uno de ellos tendrá asociado un único valor entre $50, 100, 150, 200$. Es importante remarcar que todos ellos, antes de comenzar a hacer la primera de las comparaciones, se encontrarán vacíos, es decir, llenos con ceros. Por ejemplo:

- S_1 representará al valor 50
- S_2 representará al valor 100
- S_3 representará al valor 150
- S_4 representará al valor 200

En cada una de las nueve comparaciones que se realizan para ver a qué intervalo pertenece el ángulo, se utiliza un registro P. Este registro es el encargado de almacenar el resultado de la comparación, es decir, si el ángulo pertenece o no al intervalo a través de la utilización de la función CMPPS (encargada de comparar elementos de tipo float). En el caso en que el ángulo efectivamente pertenezca al intervalo, entonces la double word de P que se corresponde con el ángulo en cuestión será llenada de unos, en los contrario de ceros.

Luego, dependiendo de qué intervalo en particular se trate, se efectuará la función OR entre S_i y P. De esta forma, los datos ya almacenados en S_i no serán borrados y además donde P sea cero (osea que el ángulo correspondiente no pertenezca al intervalo) gracias a la funcionalidad de OR no modificará el resultado parcial obtenido hasta el momento. Una vez finalizadas las nueve comparaciones, tenemos los resultados en los registros S_i.

Sabemos que si la primera double word de uno de los S_i está rellena de unos, entonces al primer ángulo de R le corresponderá el valor asociado al S_i y además sabemos que para todos los otros S_i esta misma double word estará completa con ceros. Por último a través de la utilización de una máscara que contiene el número 50 en cada double word del registro, se procede a contruir cuatro registros los cuales sean poseedores de los valores $50, 100, 150$ y 200 en cada unas de sus double words.

Luego de usará la función $PAND$ entre cada uno de estos cuatro registros y los S_i correspondientes, con el fin de poseer efectivamente el valor correspondiente al ángulo, en lugar de unos. Una vez finalizado este proceso, se pasa a realizar la última acción que refiere a juntar los cuatro registros S_i en uno solo. Debido a que garantizamos que son disjuntos su unión formará un único registro que tendrá los valores correspondientes al ángulo según el intervalo al cual pertenezcan.

Una aclaración no menor es que para hacer cada unas de las comparaciones, como mencionamos anteriormente, es necesario realizar dos preguntas, por *menor* y por *mayor o igual* con respecto a los límites del intervalo. Luego de realizar ambas y obtener un resultado parcial de cada una, se procede a realizar la función AND entre los resultados obtenidos. Esto se hace con el fin de dejar pasar únicamente a los ángulos que cumplan con ambas condiciones (la cuál conllevan a una única condición de pertenencia).

Luego, una vez obtenidos los cuatro registros en donde estarán escritos los valores a los cuales representa cada ángulo, se procede a convertirlos de punto flotante a enteros y luego cambiar su formato de double word a word y posteriormente de word a byte. Por último se procede a escribirlos en la imagen destino de la misma forma en la cual escribimos en el filtro Sobel.

Función Non Maximum Supression (nonMaxSup_asm)

En el caso de esta función, como se deben hacer comparaciones con todos los píxeles vecinos, esto sólo puede hacerse para los píxeles centrales, es decir $B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $B7$, $B8$, $B9$, $B10$, $B11$, $B12$, $B13$ y $B14$ en la Figura 32.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15

Figura 32: Posiciones con las que trabaja nonMaxSup_asm.

La función compara primero las direcciones de los ángulos. Para ello se utiliza la función `PCMPEQB` que al comparar de a bytes dos registros XMM deja 1's en el resultado si los números son iguales. Así, al comparar con cada uno de los valores asignados a cada dirección (50, 100, 150 y 200) se aplica una máscara dejando pasar solamente a los bytes de valores de interés.

Explicaremos lo que sucede con la dirección horizontal (que se corresponde con el valor 50) ya que para las demás se opera de manera análoga. En pocas palabras, luego de aplicar la máscara se debe comparar a el módulo del gradiente de cada píxel con los que están a su izquierda y a su derecha; así, si es mayor a ambos, entonces ese píxel debe ser considerado como máximo. Para lograr esta comparación se realizan dos shifts de 1 byte sobre el registro: uno a izquierda y otro a derecha para comparar luego con la función `PCMPGTB` al píxel a derecha y a izquierda respectivamente. Luego, esos resultados son almacenados.

Se repite esto mismo para cada valor correspondiente a las otras direcciones de modo que luego de eso se tienen únicamente los valores que son máximos respecto de las direcciones de crecimiento. Resta entonces hacer la comparación con los dos valores de umbral: `THRESHOLD_HIGH` y `THRESHOLD_LOW` que se encuentran almacenados en dos registros XMM que contienen dichos valores en cada uno de sus bytes.

Dicha comparación se hace primero entre el valor del módulo del gradiente en cada píxel con `THRESHOLD_HIGH` y en caso de dar mayor luego se hace una máscara con 200 a fin de que las posiciones de esos píxeles sean marcadas como fuertes. Luego se quitan a los mayores a 200 y se hace la comparación a mayores que `THRESHOLD_LOW` para, luego de aplicar una máscara con 100, tener como débiles a aquellos píxeles cuyo módulo de gradiente estaba entre `THRESHOLD_HIGH` y `THRESHOLD_LOW`. Luego, dicho resultado se coloca en la imagen destino.

3. Resultados

En la presente sección incluimos gráficos comparativos de los tiempos consumidos por las distintas implementaciones de los distintos filtros en lenguaje C con lenguaje Assembly. Dichas mediciones fueron realizadas sobre distitnas imágenes de diferentes tamaños. Considerando que el fin posterior de las implementaciones es aplicarlas sobre flujos de video y que los mismos tienen asociada una cierta resolución decidimos realizar las mediciones sobre imágenes de proporciones 4×3 y 16×9 ya que son las más utilizadas.

Para la resolución 4×3 fueron escogidos 11 tamaños diferentes; a saber: 640×480, 800×600, 1024×768, 1152×864, 1280×904, 1400×1050, 1600×1200, 2048×1536, 3200×2400, 4000×3000, 6400×3200.

Para la resolución 16×9 fueron escogidos 10 tamaños diferentes; a saber: 852×480, 1024×576, 1280×720, 1366×768, 1600×900, 1920×1080, 2560×1440, 3840×2160, 4520×2540, 7680×4320.

Para las mediciones de C y Assembly de los filtros Roberts Cross, Sobel y Prewitt se toma como valor de cantidad de ciclos igual al promedio sobre la cantidad de ciclos consumidos sobre 1000 repeticiones de la corrida (en cada imagen). Esto se realiza a fin de disminuir el impacto sobre la medición que pueda ser causado por otras operaciones realizadas por el procesador durante la medición de la corrida. Para el caso de Canny la cantidad de repeticiones es 100 debido a que el consumo de tiempo de una corrida es mucho mayor y en consecuencia el costo relativo de otras eventuales operaciones resulta menos representativo.

Todas las corridas fueron realizadas con un procesador Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz cuyos cores tienen un clock de 1200 MHz

En todos los gráficos el color verde representa las mediciones de las implementaciones en C y el color rojo representa las mediciones de las implementaciones en Assembly.

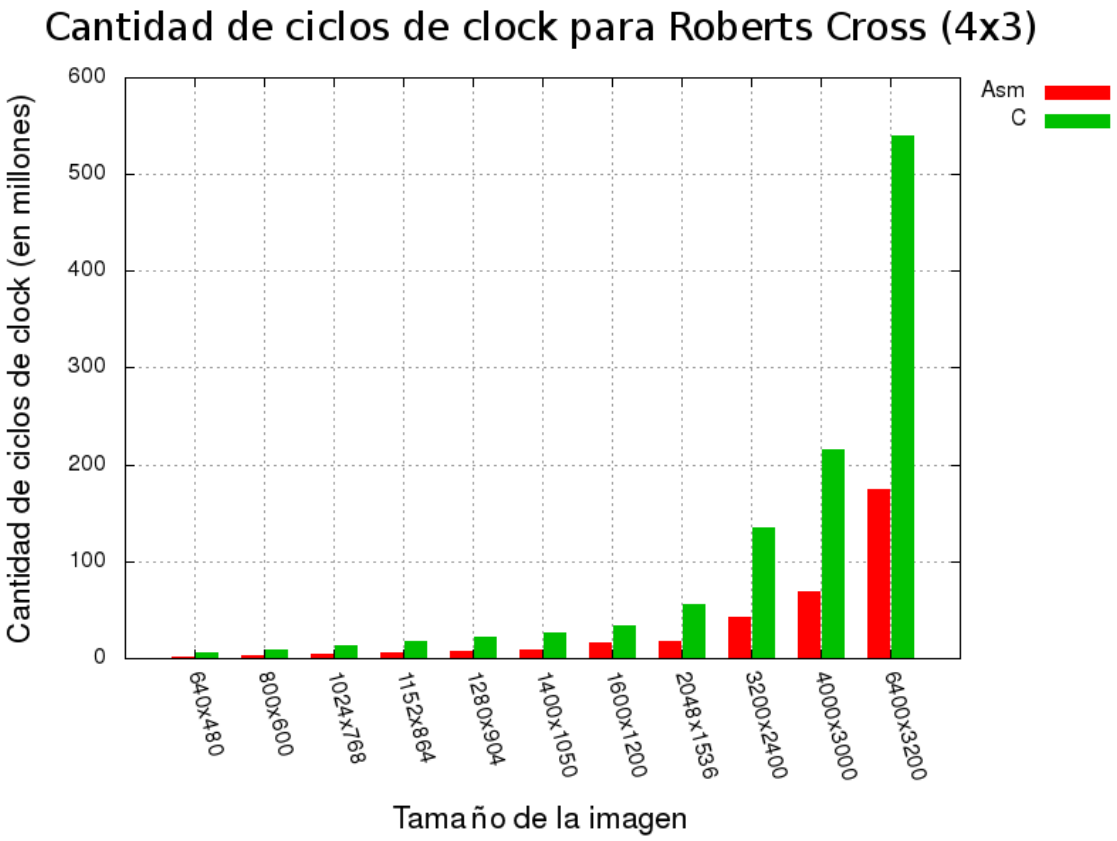


Figura 33: Mediciones para el filtro de Roberts Cross sobre las imágenes de resolución 4×3.

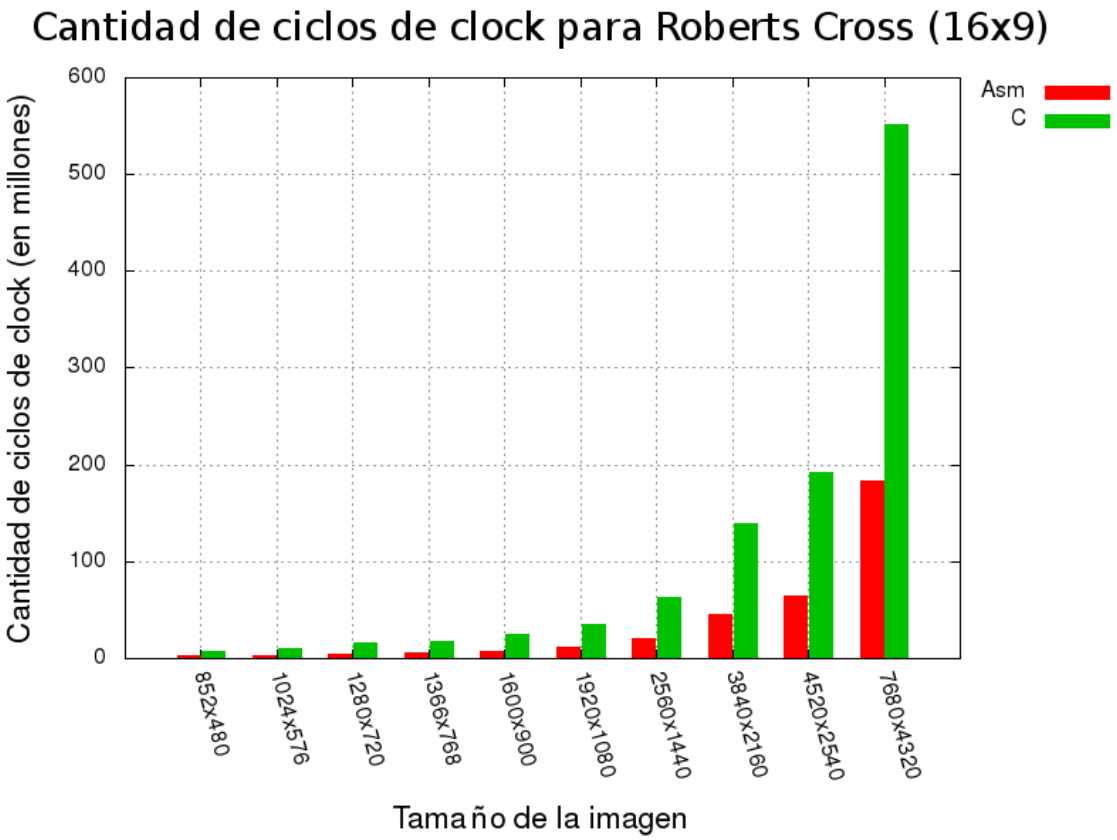


Figura 34: Mediciones para el filtro de Roberts Cross sobre las imágenes de resolución 16×9.

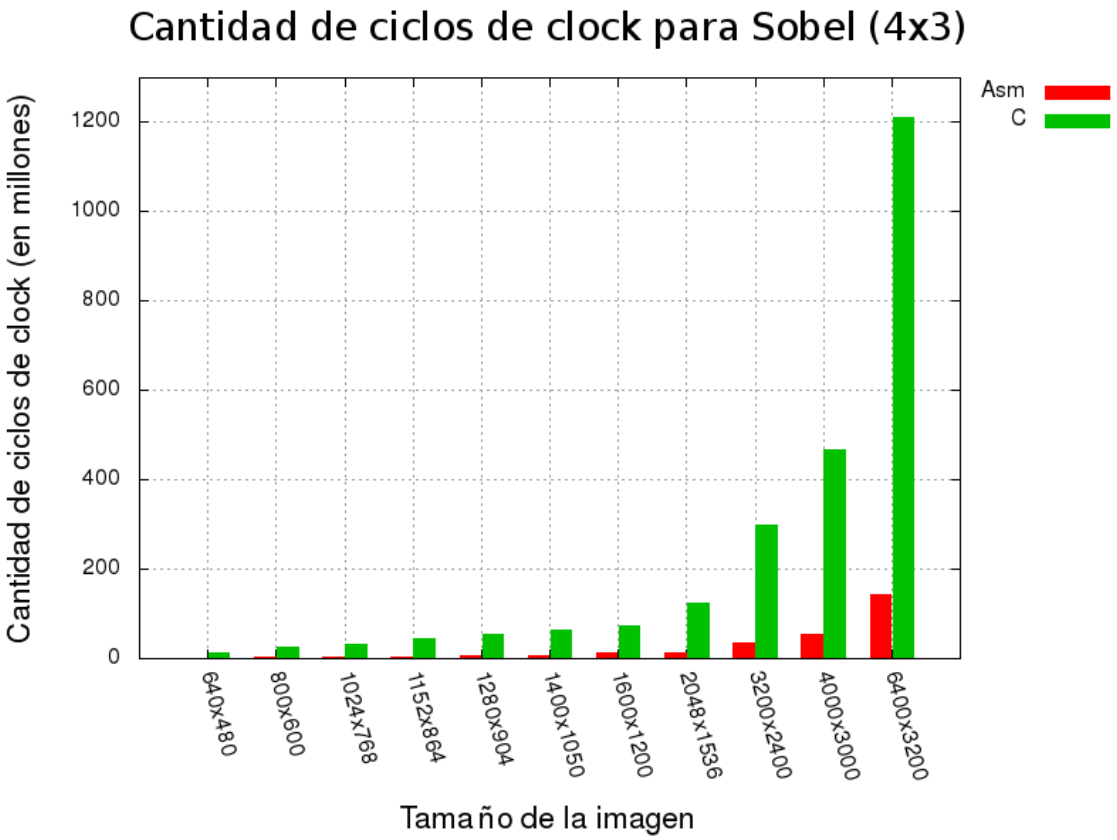


Figura 35: Mediciones para el filtro de Sobel sobre las imágenes de resolución 4×3.

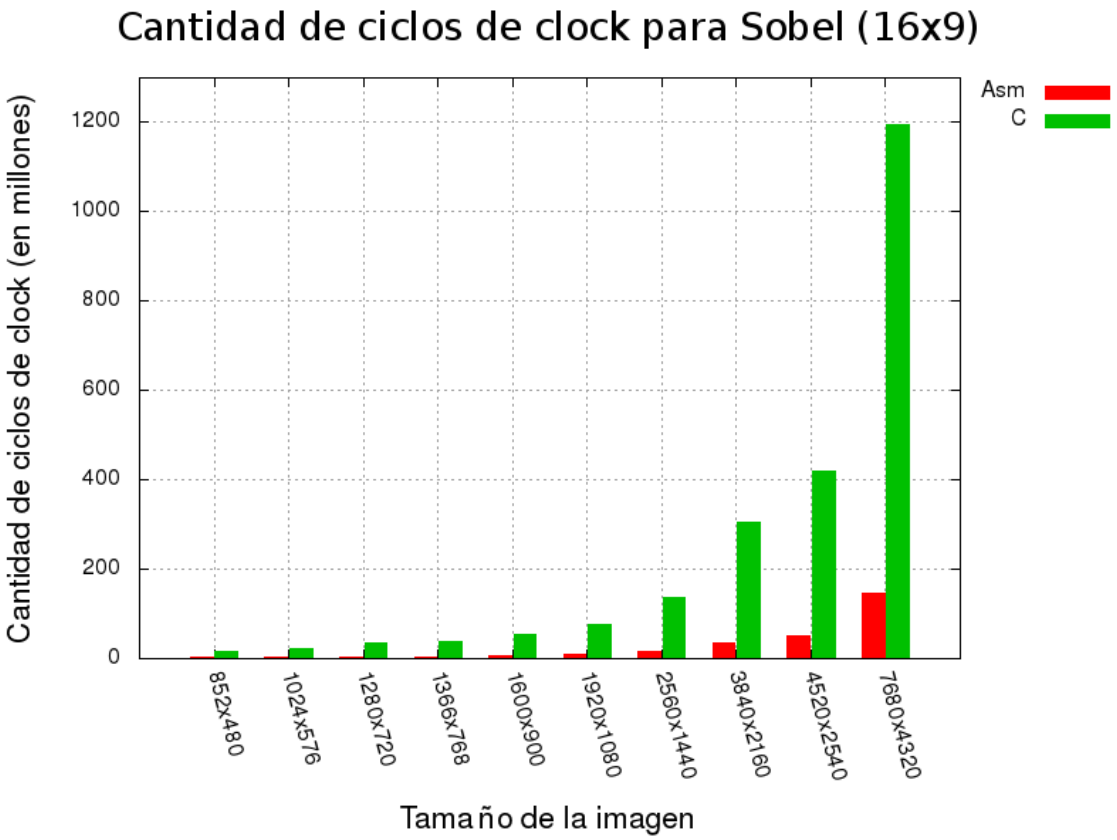


Figura 36: Mediciones para el filtro de Sobel sobre las imágenes de resolución 16×9.

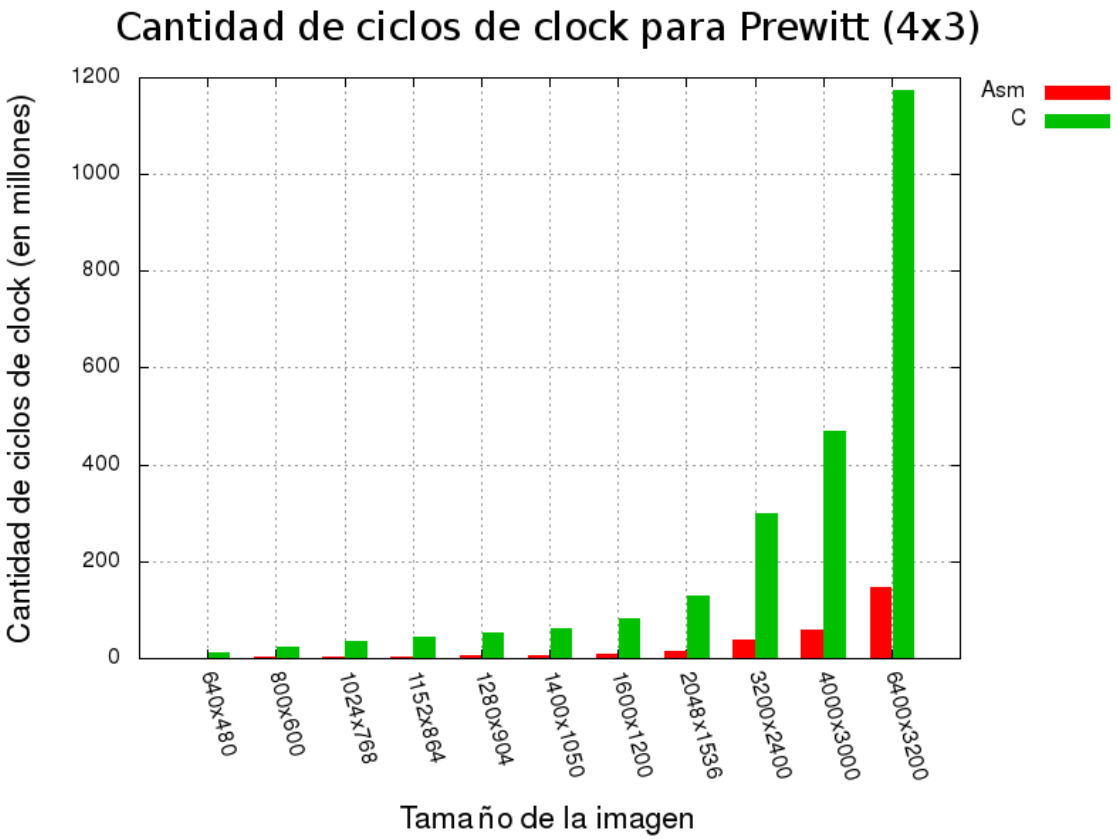


Figura 37: Mediciones para el filtro de Prewitt sobre las imágenes de resolución 4×3.

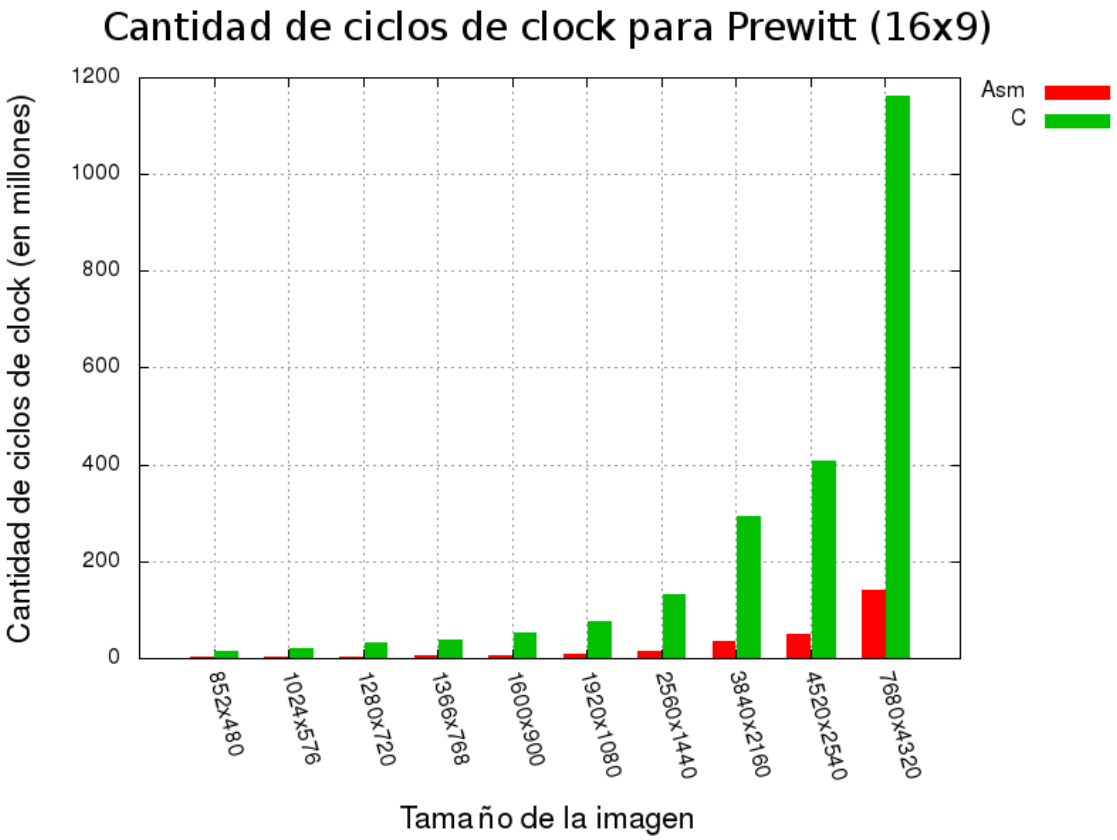


Figura 38: Mediciones para el filtro de Prewitt sobre las imágenes de resolución 16×9.

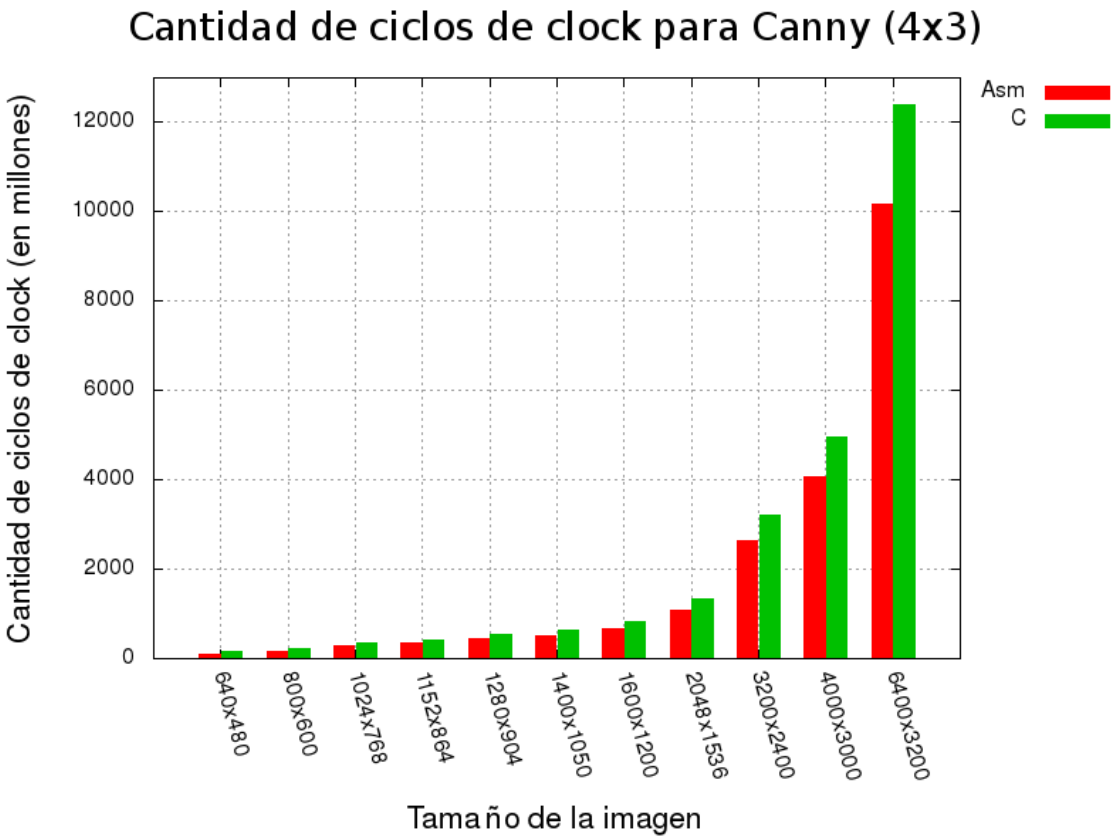


Figura 39: Mediciones para el filtro de Canny sobre las imágenes de resolución 4×3.

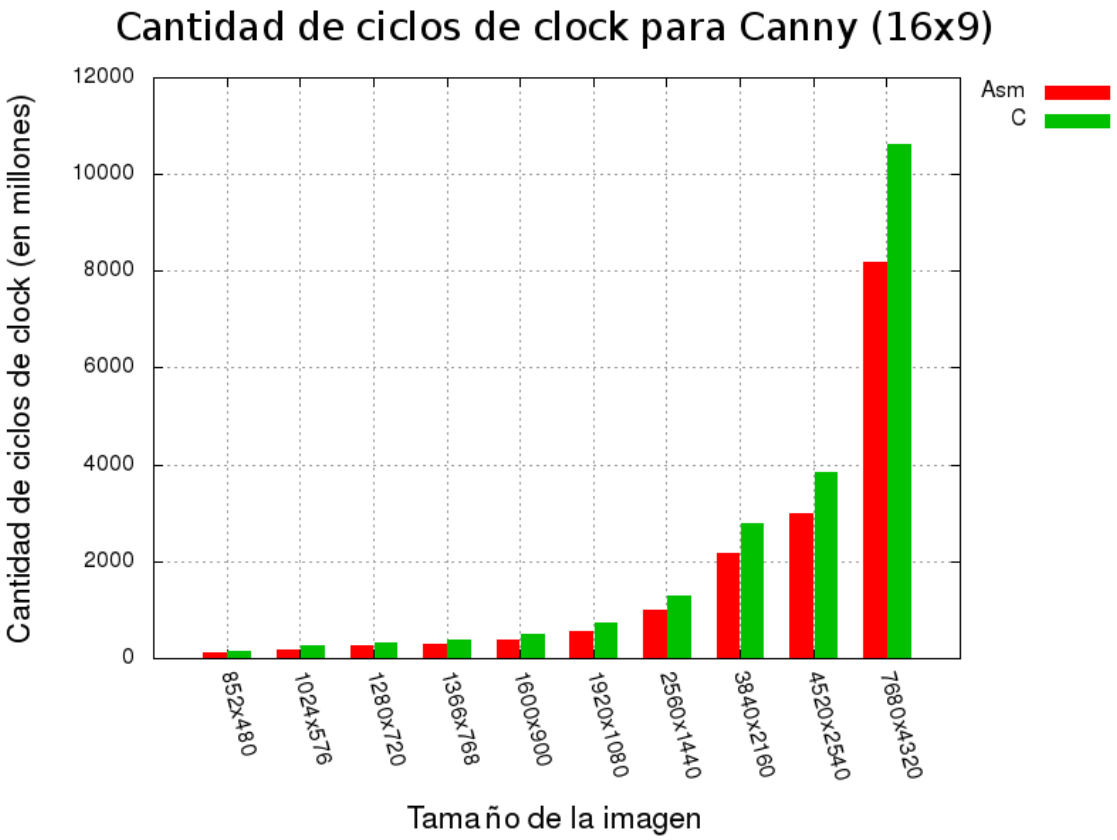


Figura 40: Mediciones para el filtro de Canny sobre las imágenes de resolución 16×9.

A continuación presentamos los gráficos en los que se comparan los tiempos consumidos por cada una de las partes del operador Canny tanto para la implementación en C como aquella en Assembly. Vale aclarar que no se presentan mediciones para la última parte (Double Thresholding) pues la misma sólo fue implementada en C++ por las razones antes explicadas.

Por último, se presentan dos gráficos más en los que se expresa el tiempo consumido por cada una de las partes sobre el total en los que sí se incluye el tiempo de la última parte.

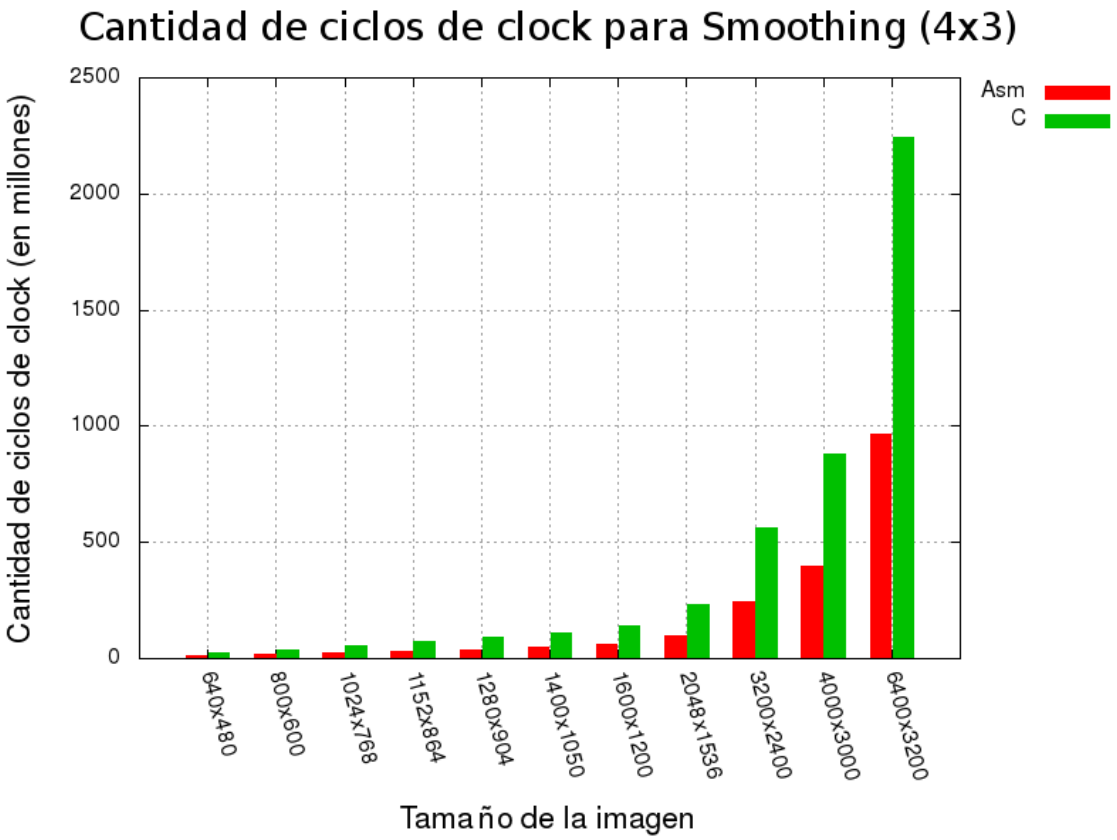


Figura 41: Mediciones para la etapa Smoothing de Canny sobre las imágenes de resolución 4×3.

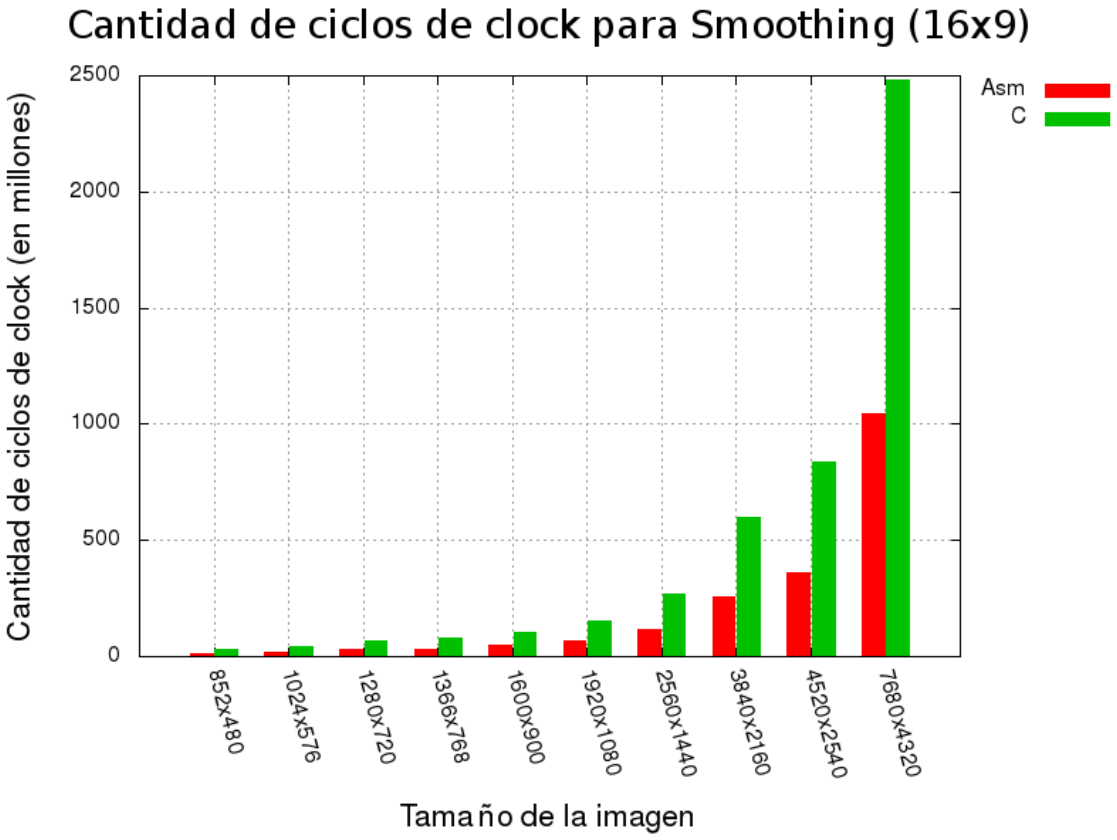


Figura 42: Mediciones para la etapa Smoothing de Canny sobre las imágenes de resolución 16×9.

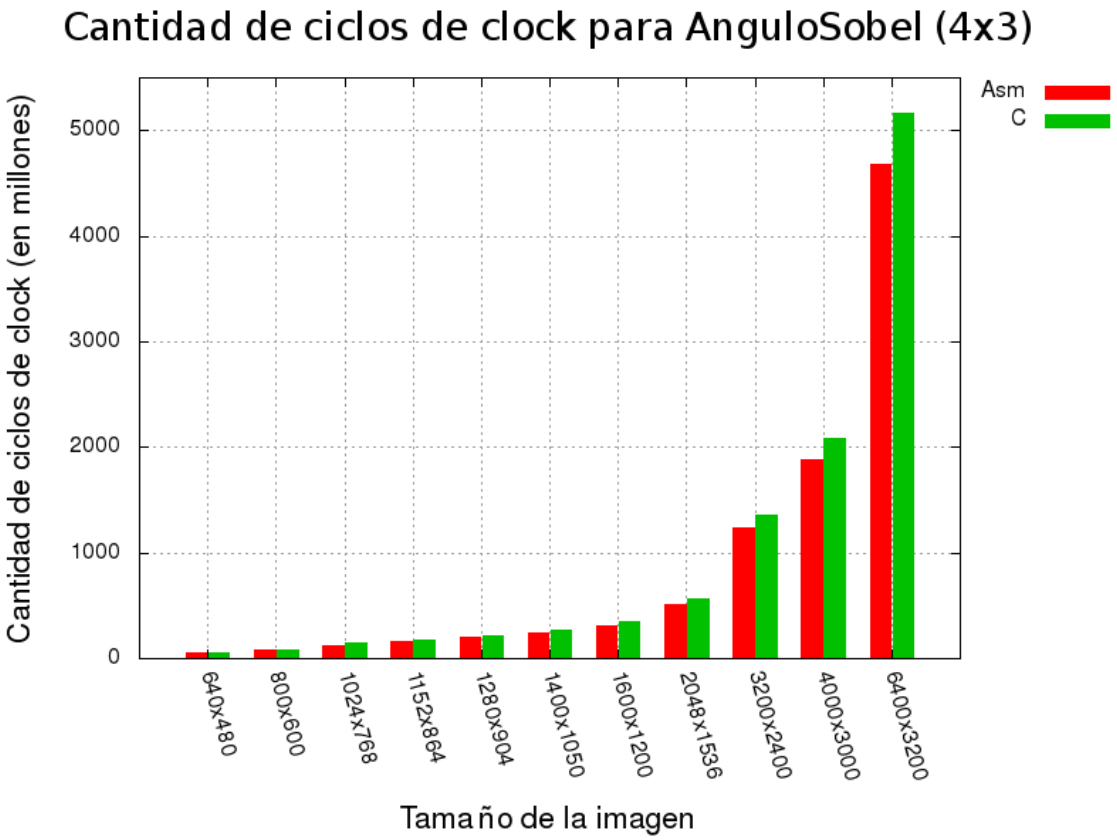


Figura 43: Mediciones para la etapa Ángulo Sobel de Canny sobre las imágenes de resolución 4×3.

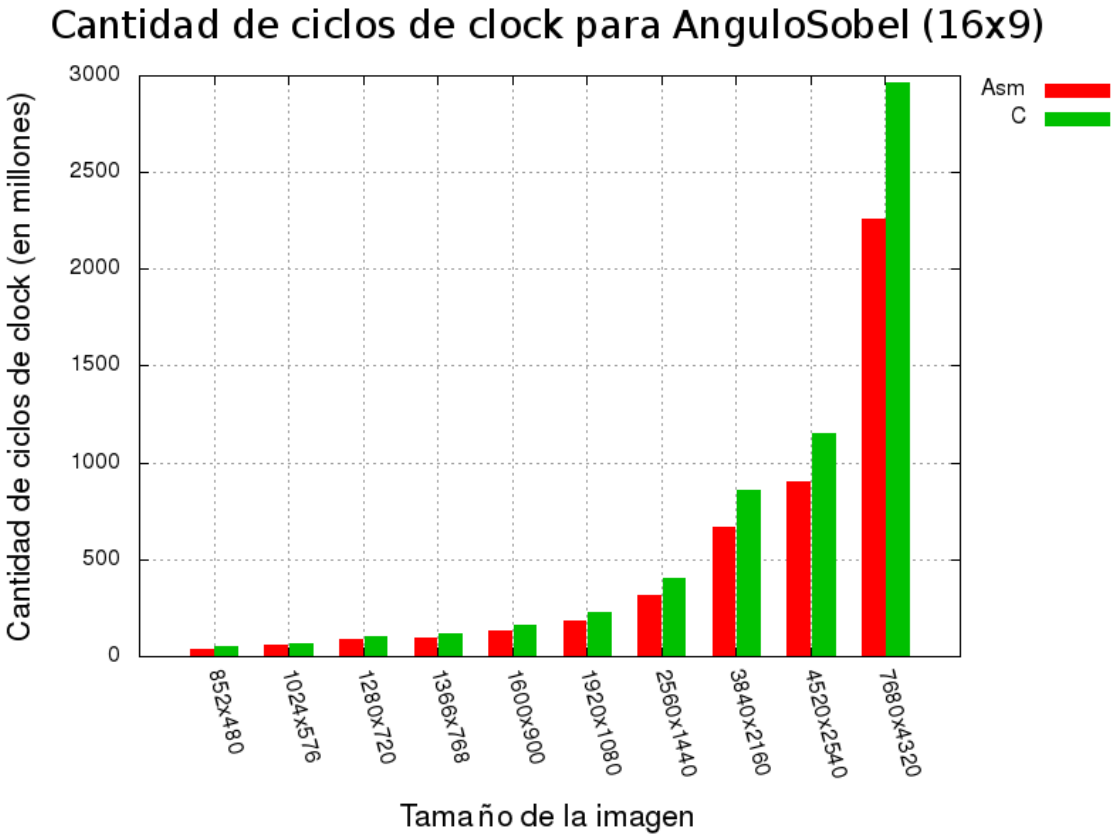


Figura 44: Mediciones para la etapa Ángulo Sobel de Canny sobre las imágenes de resolución 16×9.

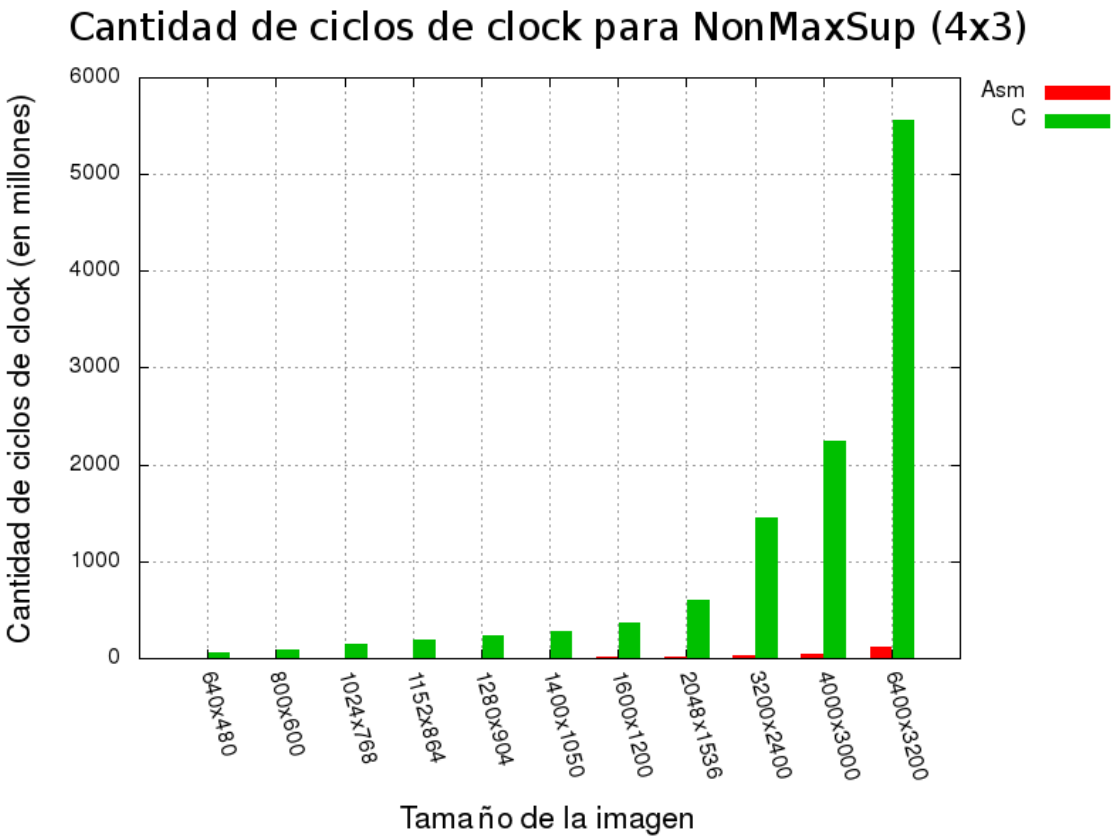


Figura 45: Mediciones para la etapa Non-Maximum Supression de Canny sobre las imágenes de resolución 4×3.

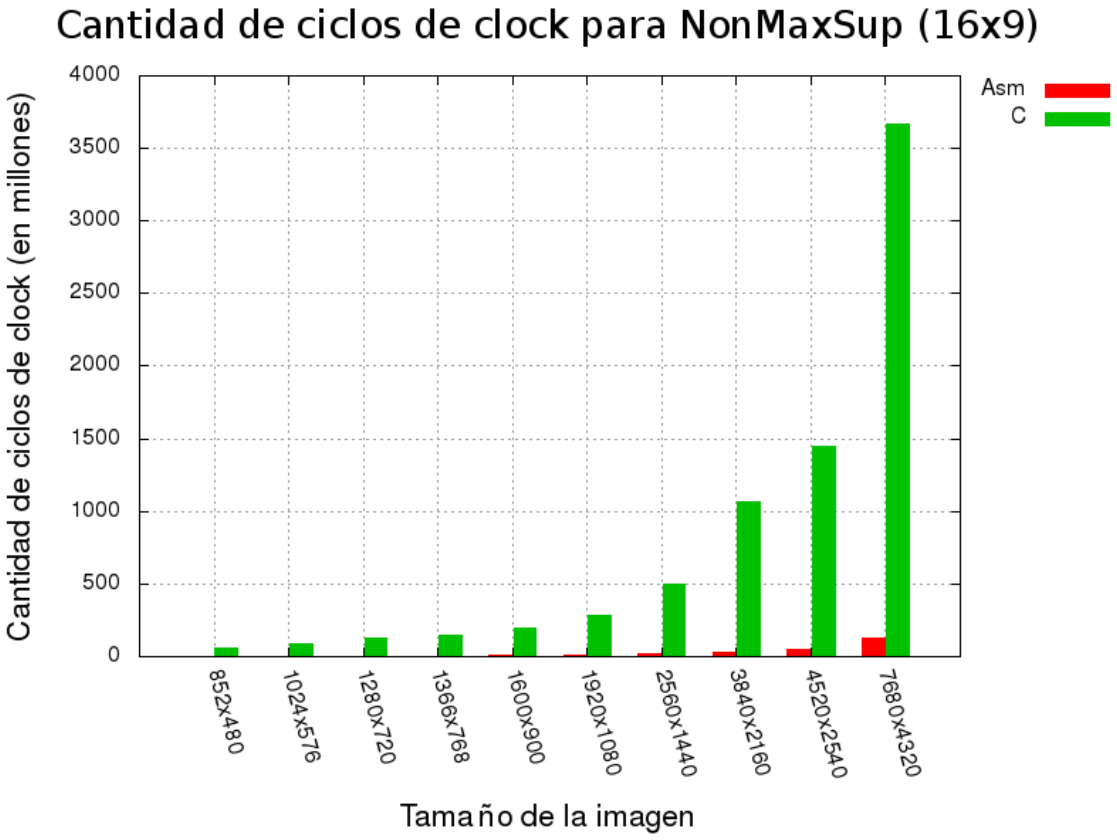


Figura 46: Mediciones para la etapa Non-Maximum Supression de Canny sobre las imágenes de resolución 16×9.

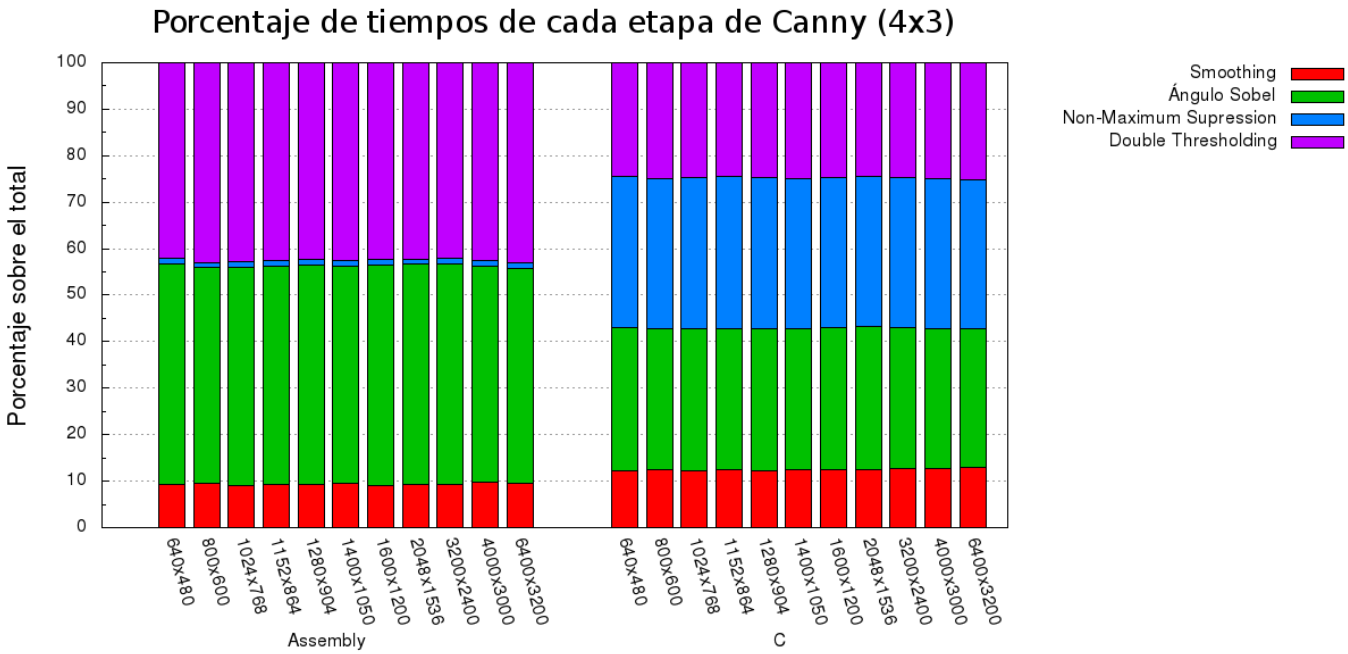


Figura 47: Mediciones para Canny expresadas porcentualmente respecto del total sobre las imágenes de resolución 4×3.

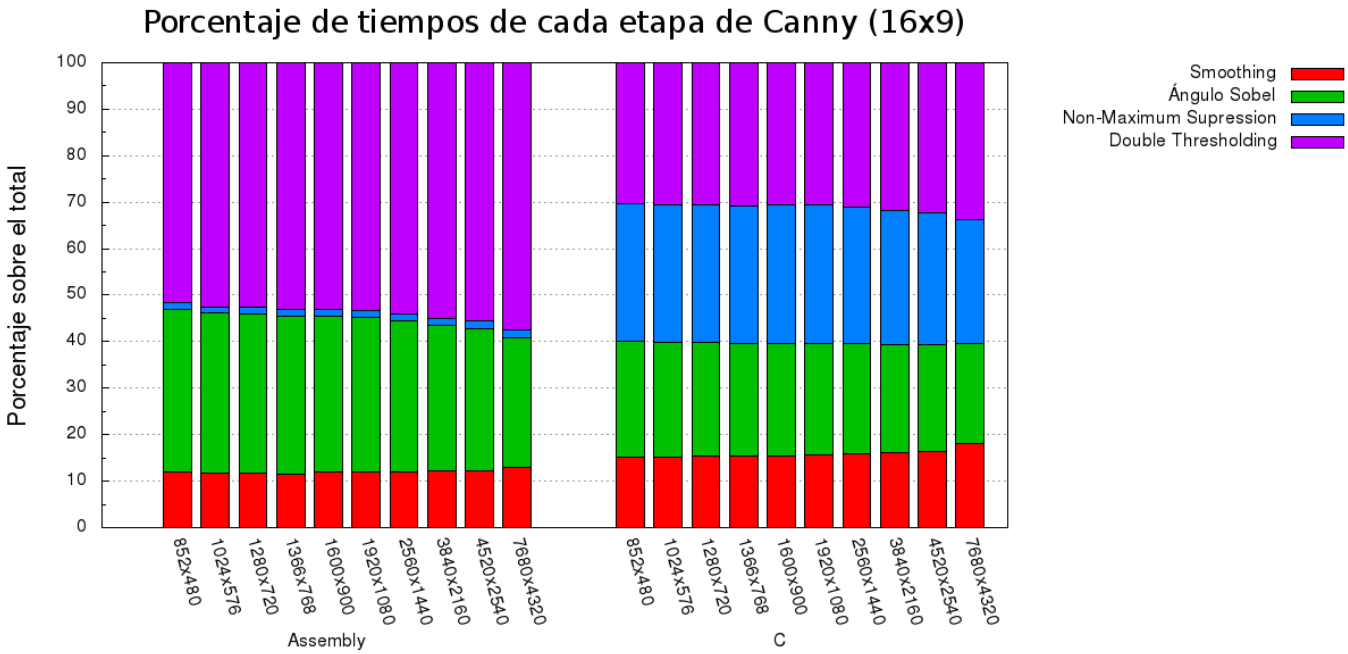


Figura 48: Mediciones para Canny expresadas porcentualmente respecto del total sobre las imágenes de resolución 16×9.

4. Conclusiones

En general, los resultados obtenidos fueron los pretendidos desde el principio. Las comparaciones entre las implementaciones de C y ASM dan mejoras en todos los casos. Los porcentajes expuestos a continuación muestran el tiempo que insume el código implementando en ASM respecto del de C según las figuras 33, 34, 35, 36, 37, 38, 39 y 40:

- Roberts Cross insume aproximadamente un 31 %
- Sobel insume aproximadamente un 11 %
- Prewitt insume aproximadamente un 10 %
- Canny insume aproximadamente un 81 %

Los primeros tres filtros dieron muy buenos resultados disminuyendo el tiempo consumido a una tercera y a una décima parte. Respecto de la diferencia de 1 % entre Sobel y Prewitt se debe a que el primero difiere en sólo algunas operaciones más debido a los valores de la matriz de convolución asociada.

Luego de observar el alto porcentaje insumido por ASM en el Canny, decidimos analizar las distintas partes que componen este filtro y observar el por qué de estos resultados. Utilizando las mismas imágenes, medimos el tiempo de cada una de las tres partes que componen a Canny (la cuarta parte no es tenida en cuenta porque solamente está implementada en C). A continuación exponemos los porcentajes obtenidos en cada una de ellas según las figuras 41, 42, 43, 44, 45 y 46:

- Smoothing insume aproximadamente un 43 %
- Angulo Sobel insume aproximadamente un 90 %
- Non Maximum Supression insume aproximadamente un 2 %

Las conclusiones que obtuvimos fueron que las partes Smoothing y Non Maximum Supression cumplieron con nuestro objetivo. Si bien la primera de ellas redujo notablemente el tiempo, la segunda superó nuestras expectativas haciéndolo de una excelente forma.

Por el contrario, Ángulo Sobel no acompaño la mejora de las otras partes, apenas logrando una mejora del 10 %. Teniendo en cuenta que esta función utiliza el filtro Sobel que dió buenos resultados, resulta más llamativo el alto consumo de tiempo. Esta función se encuentra separada en tres partes: filtro Sobel, ecuación de Pitágoras y Discretización del valor del Ángulo. Como dijimos, la primera presenta una mejora notable (del 90 %). La tercer parte utiliza comparaciones entre números utilizando instrucciones de SIMD que permiten trabajar en simultáneo a fin de consumir menor tiempo con respecto a C. Se deduce de acá que la segunda parte es la problemática. Ésta trabaja de a un pixel a la vez utilizando la FPU pues no se puede calcular arco tangentes con instrucciones SIMD. Creemos que la utilización de ella y el trabajo unitario con píxeles repercuten fuertemente en el tiempo final del algoritmo.

Luego de obtener estos resultados y teniendo en cuenta que Double Thresholding está implemenado sólo en C, supusimos que quizás esté ocupando mucho tiempo de cómputo. Por este motivo es que decidimos realizar una presentación distinta de los datos que nos permita observar otros factores. Sus resultados se ven reflejados en las figuras 47 y 48 donde se muestran cuánto influye cada una de las partes de Canny porcentualmente con respecto al total de tiempo consumido por el algoritmo.

Estos resultados muestran y confirman nuestras sospechas sobre la perfomance de la parte de Ángulo Sobel y de Non Maximum Supression. En ambas figuras se observa cómo el costo de Non Max Supression en ASM es mucho menor en relación al tiempo total, ocurriendo lo contrario con Ángulo Sobel. En la parte de las figuras que representan a Assembly sería ideal encontrarse con que Double Thresholding representa una gran parte del total del tiempo consumido, es decir, que los filtros implementados en ASM ocupen poco tiempo. Sin embargo, aunque Smoothing y Non Maximum Supression sí cumplen con lo antes mencionado, se puede observar que ángulo Sobel representa aproximadamente el mismo tiempo que Double Thresholding.

Teniendo en cuesta esto, creemos que si existiese una instrucción de SIMD que permitiese calcular arco tangente en paralelo, entonces Ángulo Sobel podría acelerar notablemente su procesamiento y, en consecuencia, mejoraría el rendimiento global del operador Canny.

Por último haciendo una conclusión general, haremos comparaciones entre los resultados de los filtros y los tiempos que tardan. Dado que Prewitt y Sobel tardan aproximadamente el mismo tiempo, cuando diremos Sobel en realidad nos estaremos refiriendo a los tiempos de ambos.

Si bien en C Roberts Cross consume aproximadamente la mitad del tiempo que Sobel, las implementaciones en ASM no mantienen esa relación sino que Sobel es más rápido que el primero. Si tenemos en cuenta las calidades de los resultados que ofrecen ambos filtros, Sobel (y Prewitt) resultan ser mejores. Considerando estos dos criterios, podemos afirmar que Sobel y Prewitt son muy superiores a Roberts Cross considerando las implementaciones en ASM.

Respecto de Canny se puede argumentar que la implementación de C supera al tiempo de Sobel/Prewitt (más aún Roberts Cross) en un orden de diez veces. Sin embargo, la implementación de ASM muestra una diferencia aún mayor (de aproximadamente 100 veces). Sabiendo que la perfomance a nivel calidad del resultado de este filtro es mejor comparado con las anteriores (previa elección de umbrales óptimos) pero que el tiempo que conlleva aplicarlo es extremendamente superior al de los demás, no se puede definir a priori cúal de ellos convendría utilizar. Creemos que depende fuertemente del contexto. En el caso en que uno busque calidad y posea tiempo para ello, lo ideal sería la utilización de Canny (ajustando correctamente los umbrales). Caso contrario, donde no se busque quizás la mejor calidad posible pero sí obtener un resultado en un tiempo relativamente pequeño, creemos que Sobel/Prewitt sería la mejor opción.

5. Interfaz gráfica

La segunda etapa del trabajo consistió en desarrollar una interfaz gráfica capaz de aplicar los filtros desarrollados en las secciones anteriores. La misma permite aplicarlos sobre imágenes, videos y capturas provenientes de webcams y realizar mediciones en términos de distancias previa configuración de una escala.

El programa puede ser compilado en procesadores de 64 bits así como también en aquellos de 32 bits. Sin embargo, dado que las funciones implementadas en Assembly han sido programadas en su versión de 64 bits, entonces en la interfaz de 32 bits sólo es posible aplicar los filtros implementados en lenguaje C.

En la siguiente subsección se encuentra un apartado donde se explica cómo compilar la interfaz gráfica por lo que aquí no haremos hincapié en ese aspecto. A continuación presentamos posibles aplicaciones de el programa presentado.

5.1. Aplicaciones

5.1.1. Rasgos faciales

Una de las posibilidades que ofrece el trabajo es la de medición de rasgos faciales. Esta tarea se ve facilitada gracias a los filtros de reconocimiento de bordes ya que es posible detectar los rasgos con mayor facilidad. En la Figura 49 se puede observar un ejemplo donde se setea una escala para una imagen y luego se realizan mediciones sobre la cara.

La idea de esta aplicación es la de determinar distancias en los rasgos de una persona que puedan ser utilizados para control policial o para generar perfiles físicos en bases de datos policiales o documentación personal.

Otra posible utilización del programa en relación con rasgos faciales es la de permitir a cirujanos plásticos realizar mediciones precisas sobre las características de un paciente evitándole al mismo cualquier tipo de incomodidad. De esta manera, tomando algunas imágenes de frente y de perfil con alguna distancia de referencia en el fondo es posible realizar todas las mediciones de los rasgos de un paciente en una etapa posterior.

(a) Imagen de una mujer con una placa en el fondo indicando longitudes en centímetros (a la izquierda) y en pies y pulgadas (a la derecha).

(b) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 80 y 60 respectivamente. Medición entre las cruces: 19,407 cm.

(c) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 80 y 60 respectivamente. Medición entre las cruces: 14,8776 cm.

(d) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 80 y 60 respectivamente. Medición entre las cruces: 7,19061 cm.

(e) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 80 y 60 respectivamente. Medición entre las cruces: 5,02697 cm.

(f) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 80 y 60 respectivamente. Medición entre las cruces: 10,6376 cm.

Figura 49: Ejemplo de utilización de la interfaz en medición de rasgos faciales.

5.1.2. Reconocimiento de caracteres

Los filtros de detección de bordes son utilizados también como parte de un algoritmo más grande en el cual se busca cierta parte de la imagen. El ejemplo más claro de esto es el reconocimiento de patentes automático a través de una imagen obtenida por una camara con el fin de, por ejemplo, aplicar multas o detectar quién sale y quién entra en un estacionamiento. Quizás la imagen a simple vista sea legible, pero si ésta posee mucho ruido podría complicar la situación para filtrar bien lo que uno busca. A continuación presentamos una serie de imágenes donde se muestra como el filtro Canny se deshace de todo el ruido dejando únicamente la información que uno pretende.

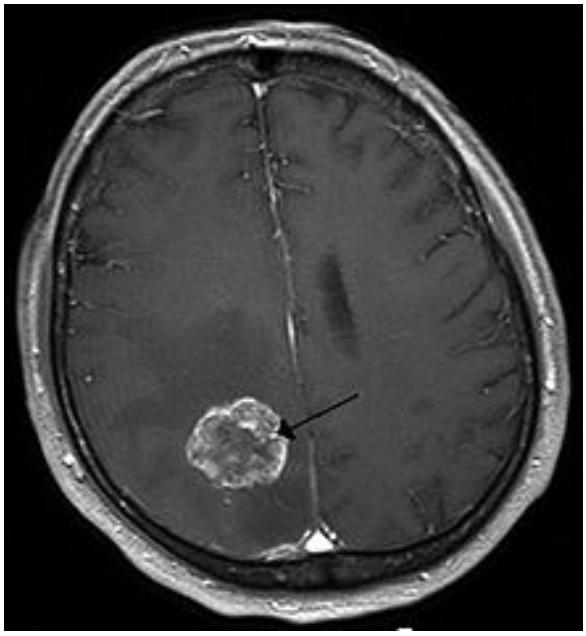
(a) Imagen original de un auto.

(b) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 135 y 75 respectivamente.

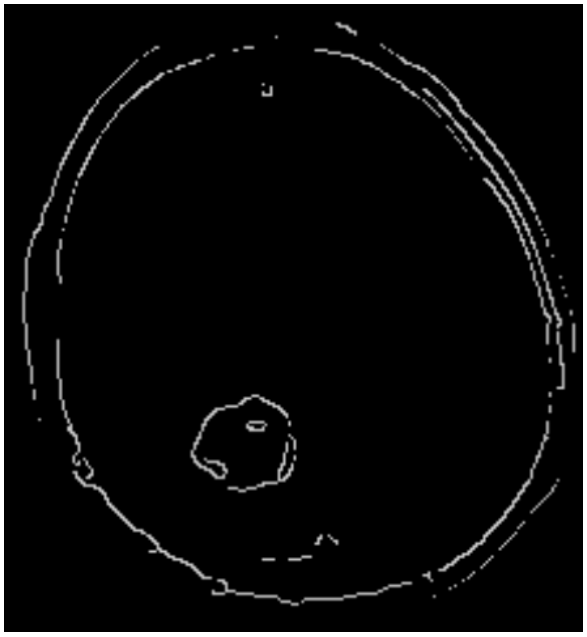
Figura 50: Ejemplo de utilización de Canny con el fin de filtrar la información no relevante.

5.1.3. Mediciones en imágenes médicas

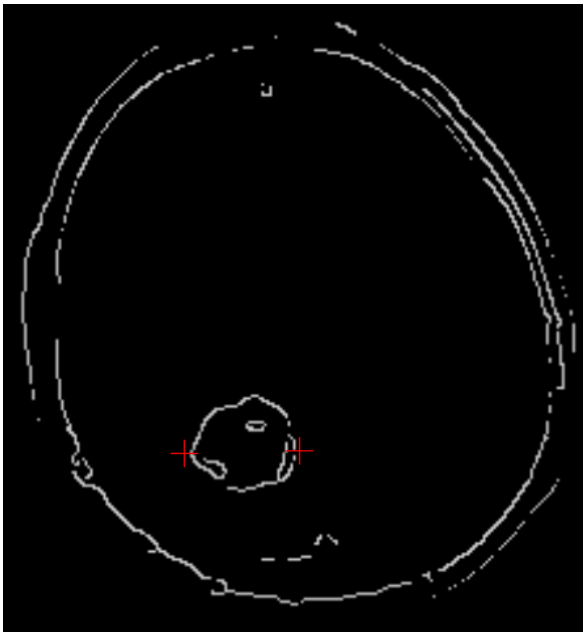
Otra posibilidad que brinda el programa es realizar mediciones sobre imágenes médicas. Luego de realizar algún tipo de estudio, muchas veces es importante realizar mediciones sobre éste para obtener datos concretos. Por ejemplo, para medir el tamaño de una fisura de algún hueso, el tamaño de un feto de una ecografía, etc. Otra posibilidad la cual ejemplificamos a continuación es una tomografía en la cual se halla un tumor cerebral. Muchas veces se quiere medir el diámetro de este, con el fin de sacar conclusiones sobre cómo seguir (saber si se puede o no postergar la operación, etc). A continuación se presenta, a modo de ejemplo, un caso en el cual medimos el tamaño de un tumor encontrado.



(a) Tomografía de un cerebro con tumor.



(b) Misma imagen luego de aplicarle el operador de Canny con valores de umbral alto y bajo 250 y 105 respectivamente.



(c) Misma imagen con una medición entre las cruces de 4,1206 cm (considerando que el tamaño del craneo era de 20 cm).

Figura 51: Ejemplo de utilización de la interfaz en medición de tumores en tomografías.

5.2. Consideraciones sobre la interfaz

Requisitos para la compilación

- Se deben tener instalados los paquetes de OpenCV [5] ya que el programa hace uso de algunas de sus funciones.
- Se deben tener instalados los paquetes de las bibliotecas gráficas de QT [6] ya que la parte gráfica ha sido desarrollada con dicho entorno gráfico .
- Para hacer uso de las operaciones programadas en Assembly es necesario tener el compilador `nasm` instalado y se debe compilar el programa en un procesador de 64 bits. De otro modo, sólo será posible aplicar los filtros en sus versiones en C.

Instrucciones de compilación

El programa viene acompañado de un archivo Makefile con las instrucciones de compilación necesarias para generar el archivo ejecutable. El mismo se encuentra en la carpeta `interfaz` por lo que para compilar se deberá situar una consola en ese directorio y ejecutar el comando `make`. Luego, para ejecutar el programa basta dirigirse al diretorio `bin` y ejecutar el binario `tpfinal`.

Características de la interfaz

La interfaz provista posee un menú con dos secciones, "Fuente" y "Opciones". Existen tres tipos de fuentes distintas desde donde pueden provenir las imágenes que serán sometidas a los distintos filtros: una imagen, un video o la captura de una webcam. Una vez seleccionada la fuente, podemos comenzar a aplicar los filtros y las funcionalidades que brinda la interfaz a través de hacer click en "Opciones".

La primera y más importante de las funcionalidades es la de elegir, qué filtro y cuál de sus implementaciones (C ó Assembler) utilizar. Esto se lleva acabo al hacer click en "Opciones" y luego en "Filtros" para seleccionar el filtro o "Lenguaje" para definir cuál de sus implementaciones. En caso de seleccionar el filtro Canny, recordar que posee parámetros que pueden ser modificados y que por default éstos poseen valores 100 y 60.

Otra funcionalidad que presenta la interfaz es la de realizar mediciones dentro de las imágenes. Estas mediciones son posibles una vez que una escala fue seteada. Para setearla se debe ir a "Opciones" y luego a "Setear escala". Una vez que aparece el campo para completar con la escala, se lo completa con el número en las unidades correspondientes y se hacen dos clicks en la imagen marcando qué distancia representa el número ingresado como escala. Luego se procede a dar click en el botón "Cargar Escala".

Una vez finalizado el proceso de setear la escala, hacer mediciones resulta fácil. Simplemente se realizan dos clicks en la imagen de modo que entre ellos esté la distancia que uno quiere medir. Al realizar el segundo click, automáticamente aparecerá el resultado de la medición. Para realizar una nueva medición es necesario resetear la última medición.

La última funcionalidad, es la de poder guardar la imagen. Indiferentemente de cuál sea la fuente provista (imagen, video o webcam) es posible almacenar la imagen con el filtro seleccionado aplicada en ese momento. Al seleccionar esta opción aparecerá una ventana para elegir el directorio y nombre del archivo a guardar. Esta opción se encuentra habilitada siempre para imágenes y tanto para video como para captura de webcam sólo cuando éstas se encuentran pausadas.

Referencias

[1] L. Roberts Machine Perception of 3-D Solids, Optical and Electro-optical Information Processing, MIT Press 1965.

[2] J.M.S. Prewitt .°bject Enhancement and Extractionïn "Picture processing and Psychopictorics",Academic Press,1970

[3] Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp.540–549.

[5] http://opencv.org/

[6] http://qt-project.org/